1992-09

# NPSNET: dynamic terrain and cultured feature depiction.

Walters, Alan Keith

Monterey, California. Naval Postgraduate School

## REPORT DOCUMENTATION PAGE

| | |
|---|---|
| REPORT SECURITY CLASSIFICATION: UNCLASSIFIED | 1b. RESTRICTIVE MARKINGS |
| SECURITY CLASSIFICATION AUTHORITY | 3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited |
| DECLASSIFICATION/DOWNGRADING SCHEDULE | |
| ERFORMING ORGANIZATION REPORT NUMBER(S) | 5. MONITORING ORGANIZATION REPORT NUMBER(S) |
| NAME OF PERFORMING ORGANIZATION: mputer Science Dept. val Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS | 7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School |
| ADDRESS (City, State, and ZIP Code): nterey, CA 93943-5000 | 7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000 |
| NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER |
| ADDRESS (City, State, and ZIP Code) | 10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO. |

TITLE (Include Security Classification)
PSNET: Dynamic terrain and cultured feature depiction.

PERSONAL AUTHOR(S)
an Keith Walters

| TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT (Year, Month, Day) | 15. PAGE COUNT |
|---|---|---|---|
| aster's Thesis | FROM 06/90 TO: 09/92 | 1992, September | 80 |

SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official licy or position of the Department of Defense or the United States Government.

| COSATI CODES | | | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) |
|---|---|---|---|
| FIELD | GROUP | SUB-GROUP | Dynamic terrain, Berms, Craters, Bridges, Distributed Simulations. |
| | | | |
| | | | |

ABSTRACT (Continue on reverse if necessary and identify by block number)

The terrain of a battlefield is in a constant state of change. There are new berms and emplacements being built, d bombs are falling leaving a crater marked terrain behind. There are bridges that must be crossed and bridges that ay not be crossable. Dynamic terrain is currently not implemented in virtual battlefield simulators such as SIMNET d NPSNET, and as a result there is a lack of needed realism to the battlefield. This work adds the dynamic features: rms, craters and bridges into NPSNET and increases the realism of the simulator dramatically. Vehicles in the nulation realistically traverse the features, tilting and rolling as they should on bumpy terrain. This work was complished using C++ and object-oriented programming, adding tremendous flexibility and growth potential to the w terrain and its features, as well as easier maintenance for later users.

| | |
|---|---|
| DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS | 21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED |
| NAME OF RESPONSIBLE INDIVIDUAL: M. J. Zyda | 22b. TELEPHONE (Include Area Code): (408) 646-2174; 22c. OFFICE SYMBOL: CS/Zk |

FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

Approved for public release; distribution is unlimited

***NPSNET: Dynamic terrain and cultured feature depiction.***

by

*Alan Keith Walters*
*Lieutenant, United States Navy*
*B. A., Harding University, 1984*

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**

from the

**NAVAL POSTGRADUATE SCHOOL**

September 1992

Robert B. McGhee, Chairman,
Department of Computer Science

## ABSTRACT

The terrain of a battlefield is in a constant state of change. There are new berms and emplacements being built, and bombs are falling leaving a crater marked terrain behind. There are bridges that must be crossed and bridges that may not be crossable. Dynamic terrain is currently not implemented in virtual battlefield simulators such as SIMNET and NPSNET, and as a result there is a lack of needed realism to the battlefield. This work adds the dynamic features: berms, craters and bridges into NPSNET and increases the realism of the simulator dramatically. Vehicles in the simulation realistically traverse the features, tilting and rolling as they should on bumpy terrain. This work was accomplished using C++ and object-oriented programming, adding tremendous flexibility and growth potential to the new terrain and its features, as well as easier maintenance for later users.

# TABLE OF CONTENTS

## ACKNOWLEDGEMENTS

The road to the completion of this research and my master's degree has been hard, challenging, and exhilarating. I am grateful to the Navy and the faculty of Naval Postgraduate School for the opportunity and education that I could not have received elsewhere. There are many people responsible for helping me get through the hard work, but I would like to acknowledge just a few of them.

First and most important person I want to thank is my God. It is His grace and blessing that I have arrived at this point in my education, career and life.

To my wife Sandy, who kept me going to the finish, when I wanted to quit. Her patience and love is greatly appreciated.

To my father and mother who continue to encourage me to reach higher and higher plateaus in my professional career. No son could ask for better parents than I have.

David R. Pratt and Dr. Michael P. Zyda, my thesis advisors, for giving me the opportunity to work for them on NPSNET. They made the work fun and enjoyable.

Captain Leonard Tharpe, U.S. Army, has been my closest friend and companion throughout my two years of education. We were the best of study partners, and without his assistance studying for some of the difficult classes would have been more intolerable.

Lieutenant Kalin P. Wilson, U.S. Navy, has been my mentor and tutor through many programming projects. Kalin provided the superb libraries that were used extensively in my thesis work.

Pacific Grove Church of Christ who kept my spirit up and my hopes high. Aikido of Monterey for the physical exercise and tension relief from the long hours of study.

# I. INTRODUCTION

## A. BACKGROUND

A battlefield is alive with action. The terrain that was once gentle rolling hills of green grass is now marred with craters from exploded munitions. Berms and emplacements have been built to provide protection and slow the advancing enemy. Terrain, in the real world, is always changing due to weather and humans, mostly humans. Humans will build a bridge to cross a river and a levee to prevent rising flood waters. When humans decide to war they will build the emplacements and berms, and then proceed to destroy these things and leave the terrain marked with craters.

This type of dynamic terrain is not currently implemented in battlefield simulators such as NPSNET [ZYDA 92], SIMNET [IEI 92] and JANUS [JANU 86]. The terrain in these simulators remains a constant from simulation initialization to completion. There is no ability to place a bridge across a river and be able to drive over it or fall off of it. When a player fires a missile and it impacts the terrain, because he or she missed the target, there is no resulting crater. Even if a simulator uses some sort of a marking for craters, such as a burnt spot, this does not affect the vehicle in any way. This lack of dynamic terrain and vehicle interaction is a deficit to the realism of the battlefield simulators. It also results in a loss of vital tactical and strategic information when planning or reviewing a battle.

The addition of dynamic terrain to NPSNET vastly improves its realism. Dynamic terrain would has visible and physical effects on the vehicles in the simulator, thus allowing the players to be hampered by craters and berms, and to improve their chances of victory by strategic placement of berms.

## B. OBJECTIVES

The terrain in NPSNET does not currently allow for dynamic run-time changes. It consists of series of squares that are divided into two triangles per square. Each corner of the square contains an elevation value for the terrain. Once the terrain is read into the

program, it remains the same until the program ends. The purpose of this work is to propose a method to implement dynamic terrain into NPSNET using an object-oriented design in C++. Using C++ and object oriented design allows each feature to handle all details that concern it, such as: creation, displaying, and interacting with the vehicles that traverse it. The result is the ability to place a dynamic feature anywhere in the simulated world without having to modify the underlying terrain or being limited within a grid square, as well as having a visible and physical effect on the vehicle that traverse it.

The proposed implementation of dynamic terrain features in NPSNET adds a greater sense of realism to the simulator. When a player fires a cannon or missile and it impacts the terrain, the result is a crater. If a player traverses the crater his/her vehicle has the effect of driving over bumpy terrain. If a vehicle is on a bridge and drives over the edge it falls off. The added realism makes battlefield planning or a battle review more accurate and effective.

## C. SCOPE

The scope of this research is to implement in NPSNET three dynamic terrain features: craters, berms, and bridges. Each of these objects is a C++ class. This research also implements vehicle traversal for each of the features. The terrain features are able to fit any grided terrain. The NPSNET terrain base has been redesigned in C++ to support the new features. The traversal of these features by vehicles is also under the scope of this research.

## D. PREVIOUS WORK

### 1. Earthworks for Distributed Simulation

Dynamic terrain, according to Latham includes: point features: bridges and buildings, line features: power lines, fences and tracks, areal: mine fields, and three dimensional: berms and craters [LATH 92]. The basic earthwork is a three-dimensional trapezoidal cross-section that is linearly interpolated between a sequence of vertices. At each vertex, the dimensions of the cross-section can be changed. If a negative height is supplied, the feature becomes a ditch instead of a berm. The ends of the earthwork are

handled by using a close approximation of the frustum of a cone. This treatment of the ends of the berms allowed Latham to use a negative height and a shared point to make craters. The user could specify the number of polygons to represent the feature.

Earthworks can have multiple segments, and if so, the end of one segment marks the beginning of another. Just as there is a special case for the ends, there is a special handling of the joints of two intersecting earthworks. At the joint of two intersecting earthworks, there is a bevel dissecting the angle of intersection. If earthworks overlap, the most recent structure's properties are used and displayed. The surface properties are given DMA-compatible codes rather than a specific color. The same is true for soil types.

## 2. Dynamic Database Modification in Distributed Simulation

There are battlefield simulators that exist on a single machine, but the ideal simulator should be networked so that there can be many players, that can be separated by great distances. Lindberg proposed a system of Protocol Data Units (PDUs) for dynamic terrain and events [LIND 92]. This proposed system provides capability for exercise-specific modifications to be made to the various databases. It also allows for support of mission rehearsal training.

With any networked system, the content of the communications must contain sufficient detail to allow a sense of commonality. For example, the color or surface properties on one system may be drastically different on another system. The communication must contain enough information for a receiving system to understand and reproduce the feature's appearance, geometry, and placement. Lindberg proposes a set of DMA/Project 2851 [DMA 86] descriptors which can be incorporated in the Dstributed Interactive Simulations (DIS) [IST 91]Standard as the basis of appearance. It is assumed that the database is from a common source. The proposed PDUs provide the capability to describe a large variety of shapes in a basic structure format (height, width, and etc.). Also included in the PDU is the ability to use time varying geometries. Lindberg states that there are two problems associated with placement of features in networked dynamic terrain.

First, each system can have a different geometric representation of the terrain. Second, continual updates or modifications to the terrain during an exercise might evolve differently between two databases. Lindberg proposes a solution of using a geometric description that is in a coordinate system independent of the local geometry and relative to the surface plane. Other issues of placement are areal features and nonareal features. Areal features such as flooded areas, burnt areas, and airfields are flat and will fill in low areas or clip high areas. Non-areal features such as roads, berms, and trenches must fit the underlying terrain. With this proposed system, the player can remove underlying features when lacing an areal feature. Non-areal features are additive. This has the effect of building over a period of time. There are five types of PDUs proposed by Lindberg:

- Dynamic Areal Feature - oil spills, bunt area, flooding and etc.
- Dynamic Feature - bridges, roads, trenches, and etc.
- Static Feature Modification - permanent structures, roads, and etc.
- Maximum Sequential ID PDU - This is occasionally sent to all of the members to notify them of the total number of changes that have been made to the database.
- Dynamic Playback request PDU - a PDU sent by a host to another host in order to pickup a missing dynamic event.

The protocol of this proposed system is discussed in three cases: issuing a PDU, receiving a PDU and simulation manager.

***a. Host Issuing PDU:***

During start-up of a new host, the simulation manager sends the host its maximum sequential ID. If the ID is not zero, then the host checks its own list to ensure that it has all of its feature modifications and request those it does not have from the simulation manager. When a host sends a new PDU it first increments its maximum sequential ID and then attaches it to the modification PDU.

***b. Host Receiving PDU:***

If a host is just starting as a new player the system sets all of the variables for the maximum sequential ID of the other players to zero. If the host is resuming play after

a period of inactive time then it uses the last known value. When a PDU from another host is received, the system checks it internal value with the ID of the PDU received. If it is missing PDUs then it requests them from the sender of the original PDU.

*c. Simulation Manager:*

The simulation manager assumes the role of sending maximum sequential IDs and responding to all request PDUs for all host who have exited the simulation. In order to do this, the simulation manager must maintain complete detail of all the host activities. The simulation manager is also responsible for setting the initial simulation conditions by means of PDUs.

## 3. Virtual Bulldozer

Moshell and his team of scientist at Visual System Laboratory have been doing research in dynamic terrain with a virtual bulldozer [MOSH 92]. The terrain consists of a regular (x,y) grid of elevation post. Each post has a z value that represents the terrain elevation. This is the standard cartographic representation of terrain. The assumption is made that the terrain is a collection of square columns with sloping tops. These squares are referred to as cells.

The bulldozer was chosen because it is the principle earth moving tool of the military and commercial construction companies. When the bulldozer moves across the terrain pushing soil into other cells, unrealistic steep banks were the result when a simple volume-conserving approach was used. A simple kinematic approach was used to reduce CPU cycles and to yield plausible soil profiles. The height of the berm was determined by the amount of the terrain moved by the dozer. The profile was determined by the slumping and is a function of the berm height. The smoothing of the pushed soil was done with cardinal splines.

As the bulldozer moves over new cells, the terrain is compared to the bottom of the dozer's blade and the outer cell post in front are given the difference. The elevations at these outer post and the elevations of the adjacent existing elevations are put into a Cardinal

spline to compute the remain cell post heights in the berm area. This proposed approach allows the bulldozer to operate over terrain with varying cell sizes, because the control points of the Cardinal spline are determined by the bulldozer geometry and not the specifics of the terrain grid. This virtual bulldozer operates on a Silicon Graphics 4D/70GT at a rate of 5 to 10 cycles per second with a wire frame rendering.

Moshell's team proposed a battlefield simulator on an hypothetical image generator, MARK 1. The MARK 1 is supposed to similar to the M1 A1 tank simulator for SIMNET. The terrain features in this proposed system contain much less detail than the virtual bulldozer, but this is necessary for real time simulation. Craters in this system would consist of 15 to 50 polygons each and the number of craters per square kilometer is between 33 to 128.

## E. ORGANIZATION

Chapter II briefly covers a description of NPSNET terrain as it was before the implementation of this work. It also has a description of the vehicle's terrain traversal method.

Chapter III covers a description of berms and craters, and the issues that apply to each. For each, a description of description of appearance is given. Placement of these features on the terrain is also covered, as well as vehicle traversal of the features. Methods to modify the features during run-time are prescribed.

Chapter IV covers bridges and the special cases of vehicle traversal that apply only to bridges. A description of the appearance is given for the intact bridge and the destroyed bridge.

Chapter V covers networking of dynamic terrain. The communications requirements of the network regarding the use of dynamic terrain is discussed. The Protocol Data Units are proposed and described for the intergradation of the dynamic terrain features into a distributed simulation.

Chapter VI is the conclusion and the recommendations for future work.

# II. NPSNET

## A. GENERAL INFORMATION

NPSNET is a low cost real time battlefield and vehicle simulator. NPSNET is similar to SIMNET, a DOD large scale networked simulator [ZYDA 92]. NPSNET allows players to drive many different types of vehicles and some humorous objects such as flying cows. There can be as many as 500 different vehicles being driven in the world at one time. These vehicles can be autonomous and return fire when fired upon. There are numerous terrain objects such as bushes, trees, rocks and much more in the world. Depending on the model of workstation used, there are some special effects available such as fog and textured landscape. The simulator is networked via Ethernet, allowing several players to interact.

## B. TERRAIN LAYOUT

The terrain in NPSNET is based on the one kilometer standard of the military grid square [PRAT 92]. There are four display resolutions:125, 250, 500, and 1000 meters. The terrain is rendered as grid squares containing two triangles. Each of the corners of the grid square is an elevation post, and elevations for positions in the center of the grid square are computed by linear interpolation. Only the terrain that is within a specified distance of the driven vehicle is displayed and as the vehicle moves across the terrain a new section of terrain is read in from a file. There are currently no provisions in NPSNET for dynamic terrain. Once the terrain is read into the system, it remains the static until the end of the exercise. Vehicles traversing the terrain use a simple two point method to determine their orientation to the ground. This is accomplished by sampling a point to the right of the vehicle, and directly in front of the vehicle for the terrain elevation (Figure 1). The roll and pitch of the vehicles are determined from these two point. This method is simple, but sufficiently adequate to set the proper orientation. There is some micro terrain in the NPSNET simulator. Micro terrain includes features such as airfields, forest canopies, and swamp land. The vehicle's orientation is not affected by the micro terrain.

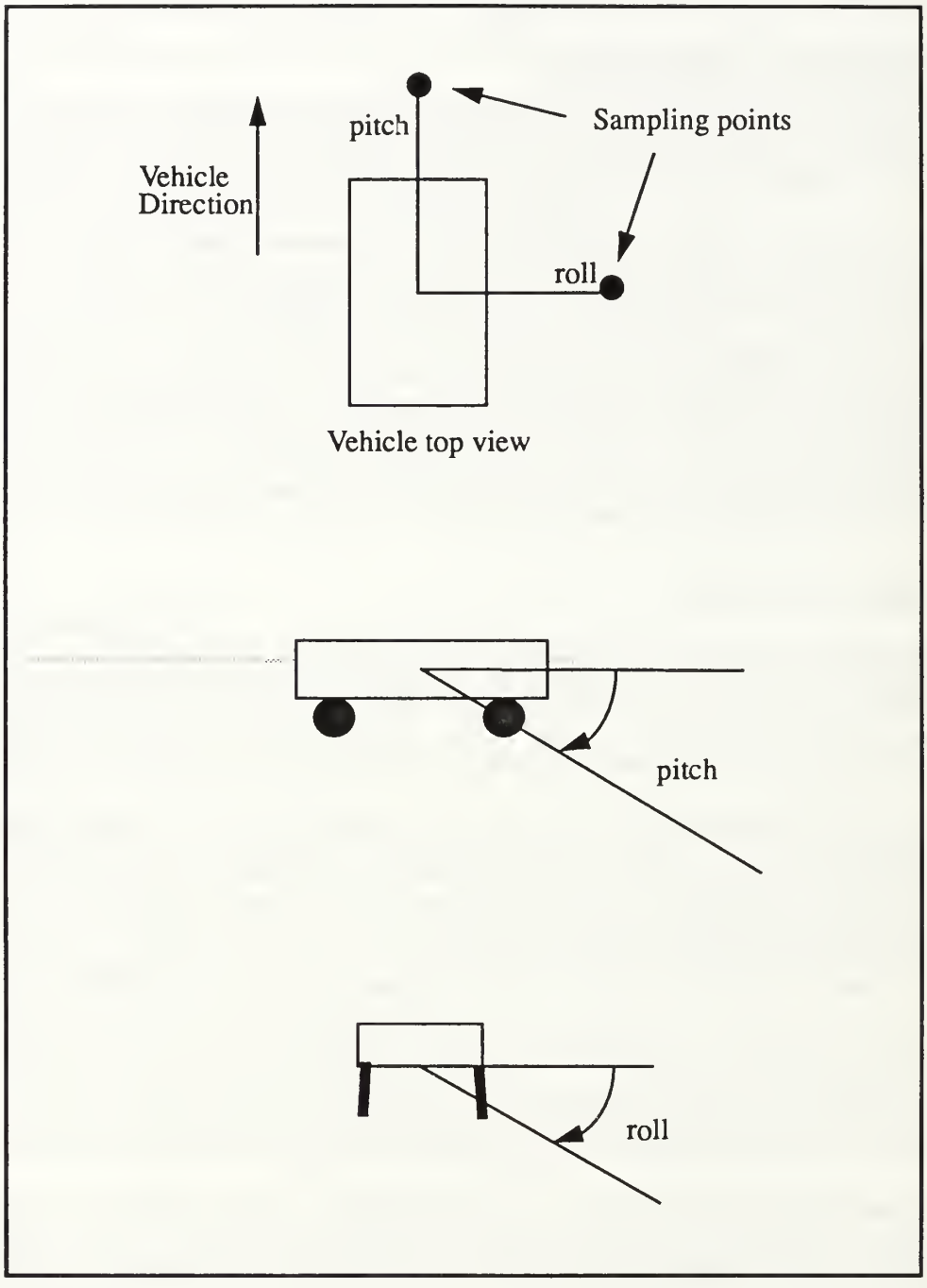



Figure 1: Two Point Vehicle Orientation

# III. EARTHWORKS

Earthworks are features that are built up by man such as: berms, emplacements, and trenches. In this research the focus is on berms, and also include craters as an earthwork.

## A. ISSUES CONCERNING PLACEMENT

In order to place an earthwork into NPSNET, an understanding of the layout and rendering of the terrain is necessary. The terrain is made up of a series of squares, with each corner of the square is an elevation. The square is divided into two triangles with the diagonal running from top left corner to the bottom right corner (Figure 2). The upper left corner is the information holder for the square. It contains information such as: the objects that are in the square, the vehicles that are in the square, and the properties of the triangles in the square. To find an elevation with in a grid square linear interpolation is used, and to find which grid square an object is in, simply divide the position values by the size of the grid. This can be done because the grid squares are uniform size. If the grid squares were of variable size, a more complicated method would have to be used.

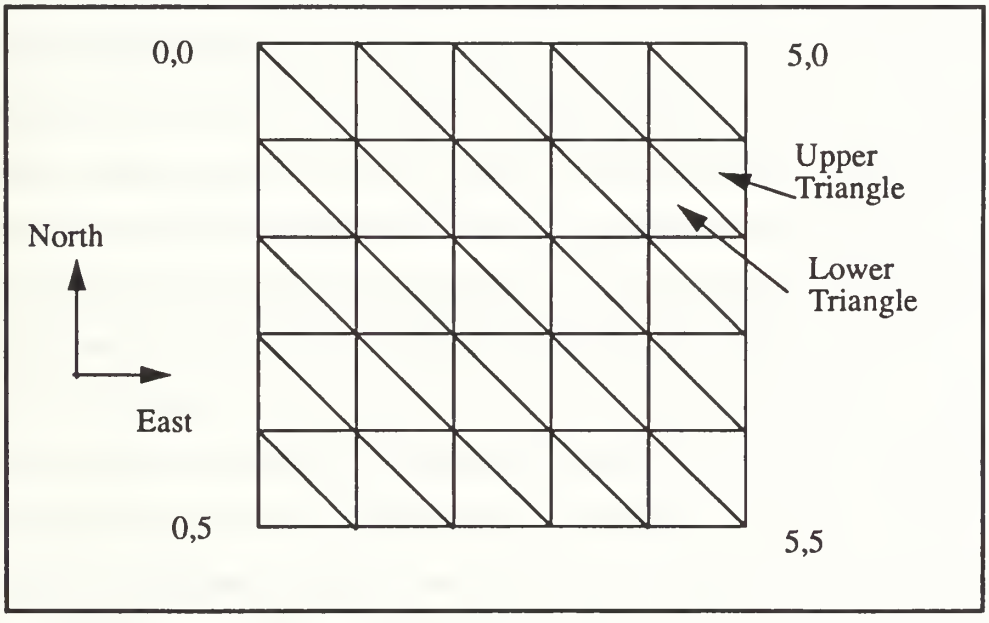



Figure 2: Underlying Terrain Grid of NPSNET

One consideration in placing earthworks into this terrain is what happens when it crosses multiple grid squares. In NPSNET, each grid square is responsible for displaying the objects that lie within its border. The objects could be limited to lie within the borders of a grid square, but that is unrealistic. If each an earthwork lies within multiple grids, then a determination must be made about which grid square is going to be responsible for pointing to, maintaining, and displaying it. The reason this is so important is because in NPSNET the terrain is rolled in from disk as needed, in order to save memory. Terrain culling is done to save time on the graphics pipeline. If only one of the grid squares pointed to the earthwork, then as a square that contains a portion of the earthwork is rolled for display, that does not control the earthwork, the result would be no interaction and no displayed earthwork. The cause of this is that the outlying grid squares do not have any knowledge of the earthwork, therefore have no reason to display or check it for an elevation.

This work made extensive use of C++'s object-oriented capabilities to design and implement the terrain and dynamic terrain features. By making each earthwork a class derived from a common base class (class deformations), virtual pointers, and reference counting pointers, each grid square can have a linked list of reference counting pointers to any earthwork that touches inside its borders. This gives the grid square access to the earthwork for interaction regardless of where the placement point is. The class itself determines if it should be displayed or if it already as been displayed, thus the grid square only has to notify the earthwork to display itself. This same approach is used for all of the access functions of the class.

The major concern of placement of earthworks is whether or not to modify the underlying terrain. To have an extremely realistic crater, this must be done. To modify the underlying for earthworks such as craters and trenches requires the addition of micro grid squares within a standard (normal) grid square or by completely changing the square layout and rendering. Changing the square layout is extremely complicated and becomes time and

computational intensive. The first might be a better approach but could also increase the amount of memory used and reduce the speed at which vehicles can move across the terrain and still be rendered real-time.

One approach to placement of earthworks is to divide each square into four equal parts and continue to do the same for each square inside of a divided square that is touched by an earthwork. In this manner the number of additional data members of the terrain database can be kept to a minimum. The approach chosen in this research was to have all craters, and berms (the earthworks covered in this research) to exist above the terrain. By using the linked list of reference counting pointers and allowing the earthworks to exist above ground, the original database does not have to be modified other than having to add a pointer to its linked list of deformations (earthworks). The following sections discuss the specifics of adding berms and craters to an existing terrain database.

### 1. Craters

Craters are usually depressions in the terrain, but also have a berm ring around the edge. They are also usually oval and irregular in shape [ARMY 83]. Having a crater that actually goes below the terrain is the most realistic approach, but that requires the terrain to be modified and drastically increases the amount of work required to access the terrain in the affected area. This research implemented the crater above ground and looking similar to a circular or ringed berm. While this is not the most realistic crater, it is still sufficiently realistic and provides greater flexibility. The craters used in the prototype were built up as a frame of four concentric circle that spread out from the placement position and sixteen spokes that originated from the placement position that are equally spaced in degrees (Figure 3). The middle two rings are the ridge at the top of the berm, and are set above the existing terrain. The craters implemented in this research are of variable radius, but are uniform in appearance (Figure 4). The construction of craters is a simple call to the class constructor, and the class constructor algorithm takes care of all that is needed to build and add the crater to the terrain (Figure 5).

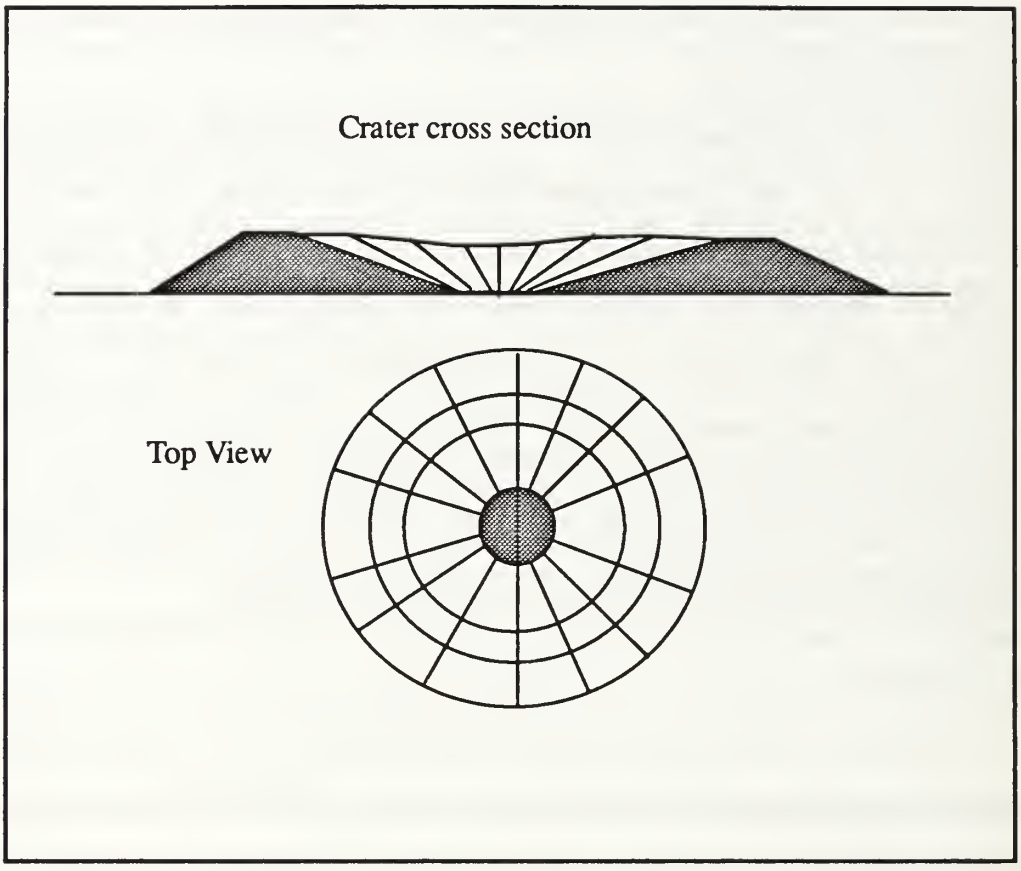



Figure 3: Crater Cross Section

Figure 4: Craters

```
crater(XXXXXXXX)
    set the placement position
    set the radius
    get the material property of the underlying terrain
    get an ID number
    for each spoke of the crater loop
        for each ring of the crater loop
            compute the x and z values for the point
            get an elevation of the underlying terrain at the point
            if the point is an inner ring
                add the offset to the elevation
            turn the indicator bit on for this area
            stack a pointer to the grid square containing this point
        end loop
    end loop
    check the stack of pointers
    for each pointer loop
        add the crater to the linked list pointed to by the pointer
    end loop
    calculate the normals for each point
end
```

Figure 5: Crater Constructor

### 2. Berms

Berms, unlike craters, do exist above ground and have no real need to modify the underlying terrain. The berms proposed and implemented consists of a starting or placement position, a direction, a length, a width, and a height. The berm length is divided into ten equal segments to allow it to follow the terrain over a hill or through a valley. If fewer segments are used, then there is a greater chance of a berm section that cuts through a mountain or floats between two hills. If the berm is too long, the same problem can occur

with the sections. Since berms can be pushed up from the ground or built with materials that were supplied such as concrete, the material for the berm can be supplied by the placer. Besides the ten sections for length there are three sections for the width. The two outer sections are the slope of the berm and the inner section is the top (Figure 6). The cross section of a berm is trapezoidal. The prototype berms in this research were designed with a standard end piece. To join two berms together, the placer needs only align the berms so that the ends intersect. Emplacements are created by joining three berms together (Figure 7). Berms are started from the placement point and runs in the specified direction and to the specified length. A different approach might be to place the starting point at the center of the berm and work in both directions, but that results in more complicated computations for construction and for finding an elevation at any point on the berm.

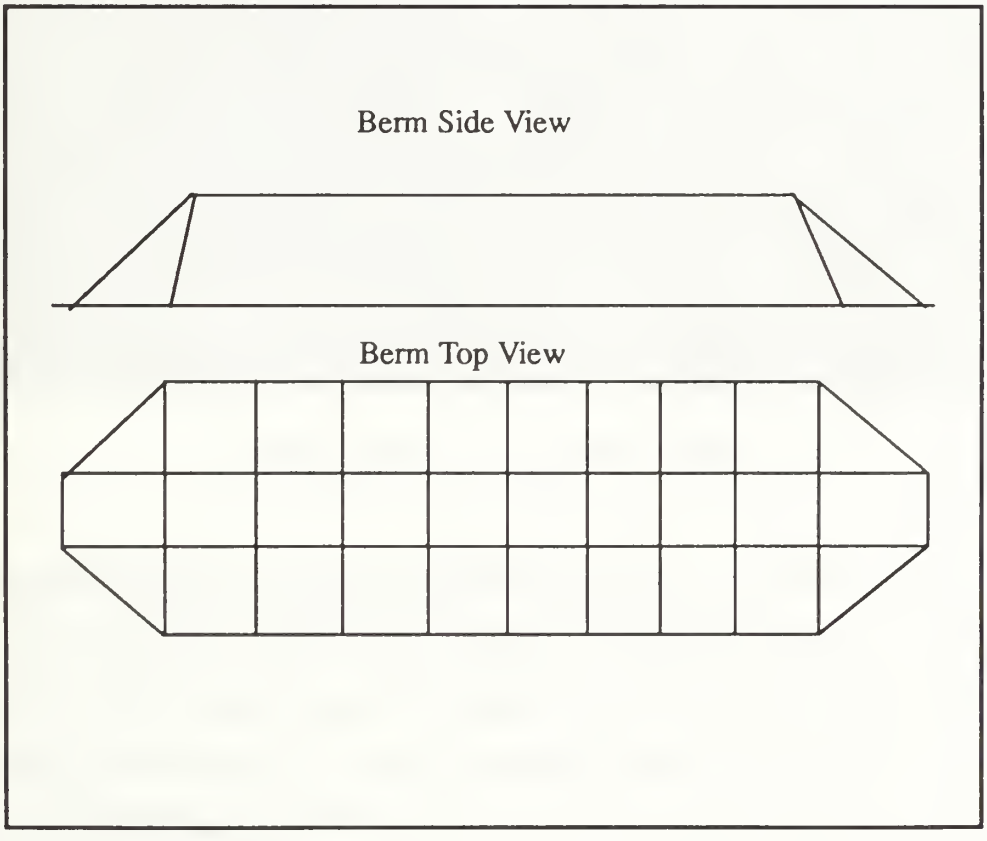



Figure 6: Berm Cross Section

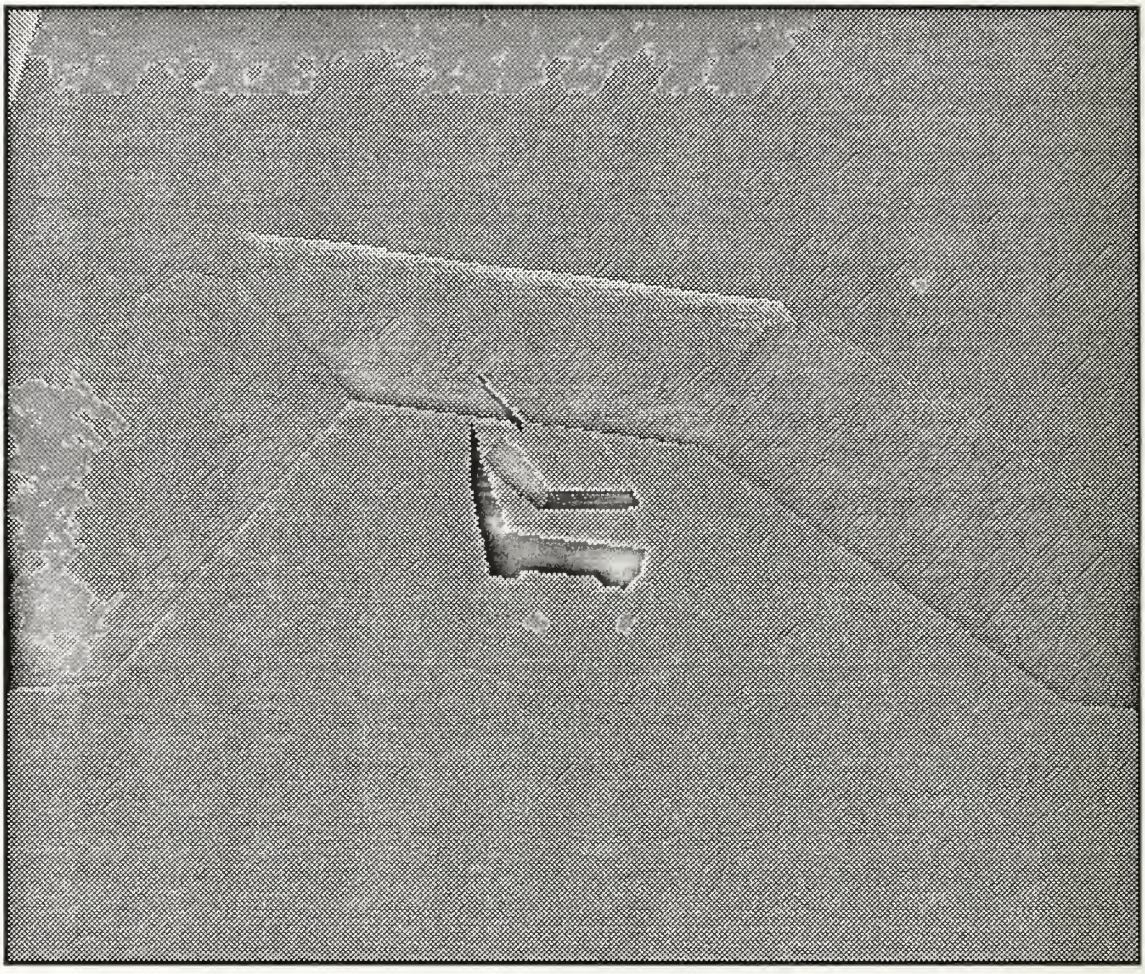


Figure 7: Berms Used as Emplacements

Growth of berms can be accomplished by changing the berm's parameters or by deleting the berm and constructing another berm which is slightly larger than the first. The first method is the simplest, because the builder need only to make calls to the parameter methods of the berm class. The disadvantage to this method is that the berm will grow from the starting point, therefore it will appear to only in one direction. If having the berm grow in both directions is needed it is best to delete the old and build a new in its place. The general berm construction algorithm is presented below (Figure 8).

```
Berm(XXXXXXXXXX)
    Set the placement position
    Set the length
    Set the width
    Set the height
    Set the direction
    Set the material
    for each width segment line loop
      for each length segment line loop
          compute the x and z value of the point
          get the terrain elevation at this point
          if the point is a top point
              add the height to the elevation
          turn the indicator bit on for the area of the point
          Stack a pointer to the grid square that the point is in
          increment the length segment
       end loop
       increment the width segment
    end loop
    check the stack of pointers
    for each pointer loop
      add the berm to the linked list of the grid square pointed to by the pointer
    end loop
    compute the normals for all of the points
  end
```

Figure 8: Berm Construction Algorithm

## B. VEHICLE TRAVERSAL OF EARTHWORKS

Having earthworks in the virtual battlefield is an extremely important part of the realism, but if the vehicles do not traverse them in a realistic manner then the earthworks are meaningless. The vehicles must be able to cross over craters and berms having the proper orientation depending on the amount of the vehicle that is on the earthwork and the location of the vehicle on the earthwork.

Determining the orientation for realistic vehicle traversal requires that the elevation of the terrain be taken in at least four, the four corners or tires of the vehicle. The method used contained six points, the four previously mentioned and two in center of each side of the vehicle (Figure 9). This is where the class hierarchy of C++ comes in handy. The vehicle needs only make a call to the terrain class for an elevation for each of the orientation points it needs. For each call to the terrain for elevation, the terrain class checks the corner point of the appropriate grid square to see if there are any earthworks. If there are earthworks present, the terrain class queries the earthwork for the elevation at the requested position. The earthwork class performs the appropriate computations and returns a value. This is done for all earthworks that fall under the requested position. The value that is ultimately to be returned to the vehicle is the greatest of these elevation values. The greatest is returned because earthworks can be overlapped and more than one may exist at the requested position. For example, a berm might intersect another slightly taller berm and it is undesirable to have vehicles drive through the taller berm because only the value of the shorter berm was returned.

Once the vehicle has the elevations it needs for each of its orientation points, it can compute the orientation. There are assumptions and issues to consider when computing the orientation of the vehicle. Previously in NPSNET the vehicle orientation was determined with only two points, one in front and one to the right side of the vehicle. This worked well as long as there were no berms or craters, but with the addition of dynamic terrain a new way of determining the orientation needed to be devised. The multipoint method mentioned above is the method chosen. The two point method is still used when there are no earthworks present. The two point method is faster than the multipoint method, therefore it should be used when there are no earthworks present. To make the determination of which method to use, the vehicle can query the terrain for the presence of any earthworks in the area where it is. This can be enhanced by using a bit grid where each bit represents a portion of the grid square. This is an extremely effective and efficient way to check the terrain, since all that is being done is checking to see if a bit has been set. The terrain, which is a

matrix of grid squares, has a bit grid large enough to divide all of the grid squares into equal parts. For example each square might be represented by a four by four matrix of bits. By using multiple bits inside of each square the vehicle only has to use the multipoint when in an area that is represented by a bit that has been turned on.

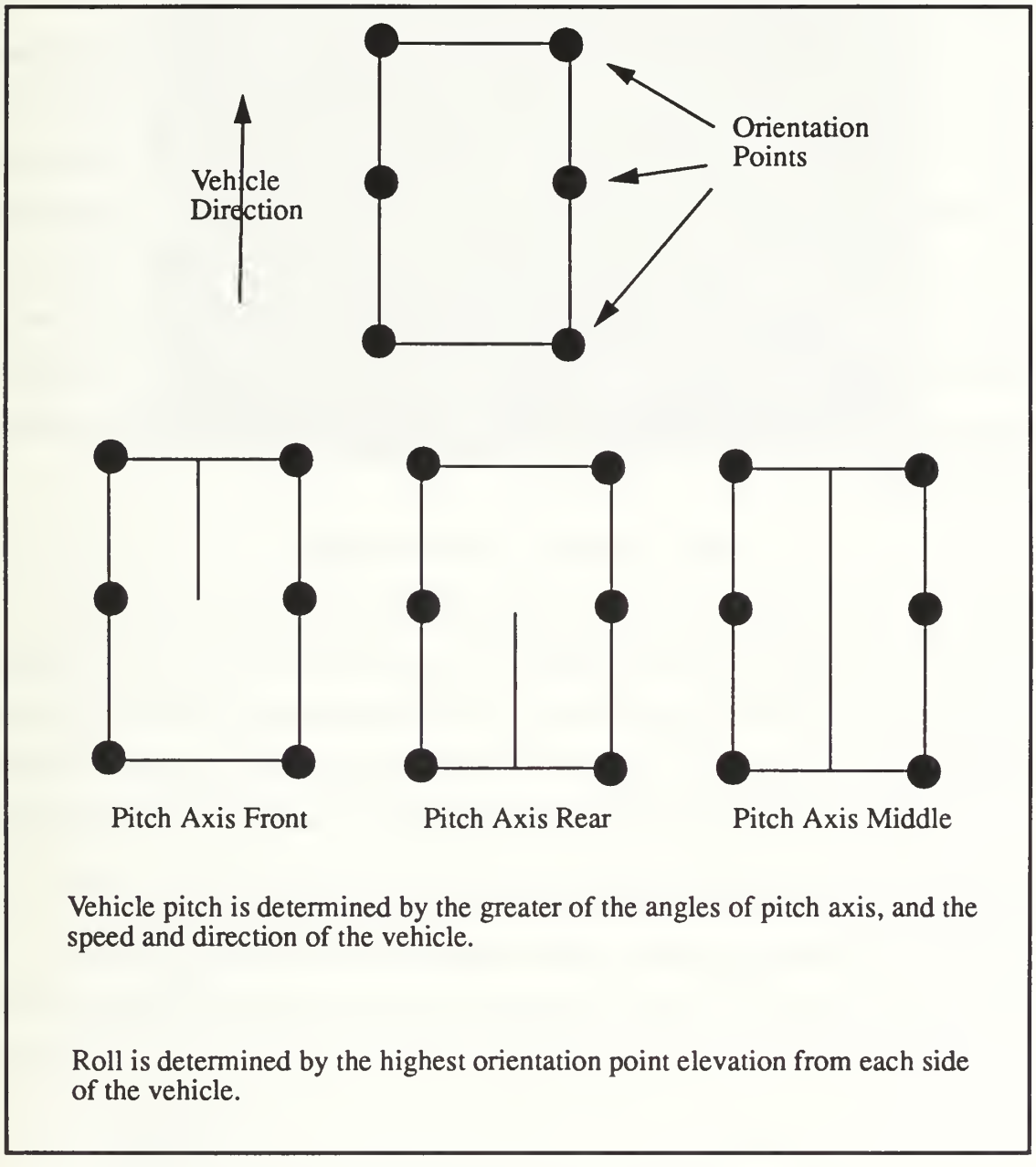



Figure 9: Multipoint Vehicle Traversal Orientation

The vehicle is considered to be a rigid body, and since the shocks of the vehicle are not modeled in NPSNET when one wheel is lifted higher, the entire side of the vehicle must be lifted (Figure 10). When computing the pitch of the vehicle, the average between the front orientation points, the center of the vehicle, the rear orientation point is taken respectively. Then based on the speed, and direction of the vehicle, the pitch is determined by taking the greater of the angles between the points. If the vehicle is traveling forward, then its speed will be positive and the angle that is greatest between the middle and the front or the rear and the front is used to compute the pitch. To compute the roll of the vehicle, the greatest elevation value on each side of the vehicle is used. The reason for this approach is for the case of a tracked vehicle. If the tracked vehicle is traversing a crater so that only one side of the vehicle contacts the crater, that side should remain at that height until the rear of the track has passed off of the crater. Speed is an important factor because if the vehicle is going backwards, it must obey the same orientation behavior as when it is going forward, only in reverse

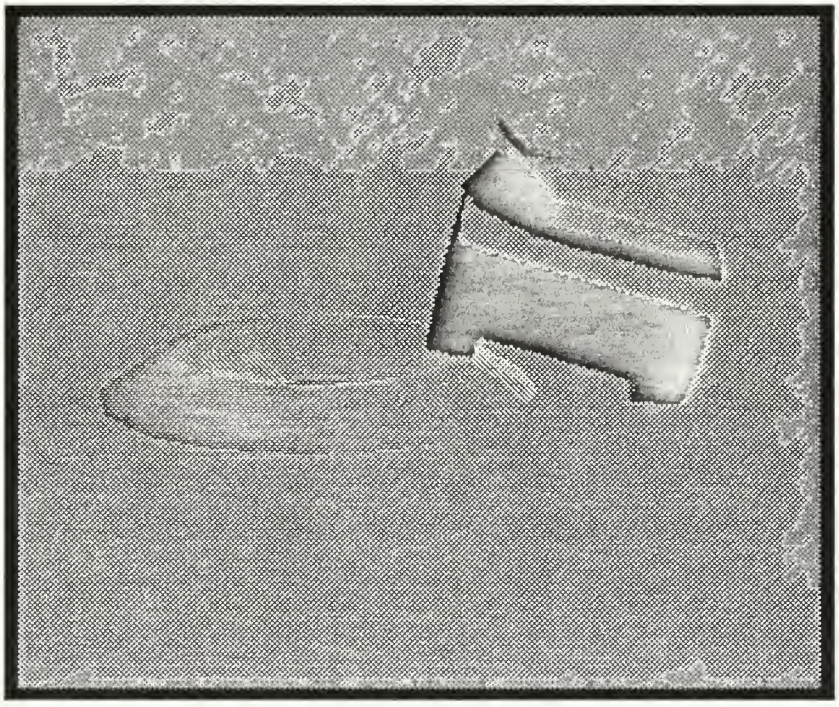


Figure 10: Tank Traversal of a Crater

Once the orientation has been computed, the elevation of the vehicle itself must be computed. If a vehicle is partially on a berm or crater, then the elevation must be computed using the orientation point elevations and not the elevation of the terrain below the vehicle center. If the elevation of the terrain at the center of the vehicle were used, the orientation based earthwork would cause part of the vehicle to disappear below the terrain.

## C. ISSUES CONCERNING RUN-TIME MODIFICATIONS OF EARTHWORKS

There are two reasons that the terrain changes, natural and man-made. This section briefly discusses the issues of change the terrain and earthworks during run time of the simulation.

### 1. Natural

Natural changes to the terrain are from sudden catastrophic events such as earthquakes, hurricanes, flash floods, and the like, or they happen over a period of time

such as erosion from wind and rain. These types of changes are not implemented due to the short duration of the simulations.

### 2. Man Made

Changes caused by man such as the building of berms and trenches, and the destruction of these features are more likely to occur within the lifetime of an exercise or simulation. When a bomb falls, the result is a crater. Berms can be breached by explosions and by machinery such as bulldozers. There are two, possibly more, methods of breaching a berm. The first is to compute the appropriate changes and set the changes to the berm or crater directly through the class access functions. The second method is to delete the original berm and replace it with the appropriate number of smaller berms. The second is less complicated to accomplish, because the same values are be used by the new berms that were used by the old berm, with the exception of the length and computing a new position for the additional berms. The second method is used in this work.

# IV. BRIDGES

## A. PLACEMENT

The placement of bridges is similar to that of the earthworks, except bridges need a more planar look and feel. Bridges, like earthworks are kept in a linked list for each grid square that is touched by it.

There are many types of bridges, but only the overpass style of bridge was implemented. The bridge has three components: two ramps and one span. The bridge can be adjusted to take on many forms by varying the parameters of the three components. Bridges have a direction, three lengths, two heights, and one standard width (Figure 11). This is all that is needed to have a realistic bridge in the simulation. This bridge is simple in form, but fairly realistic.

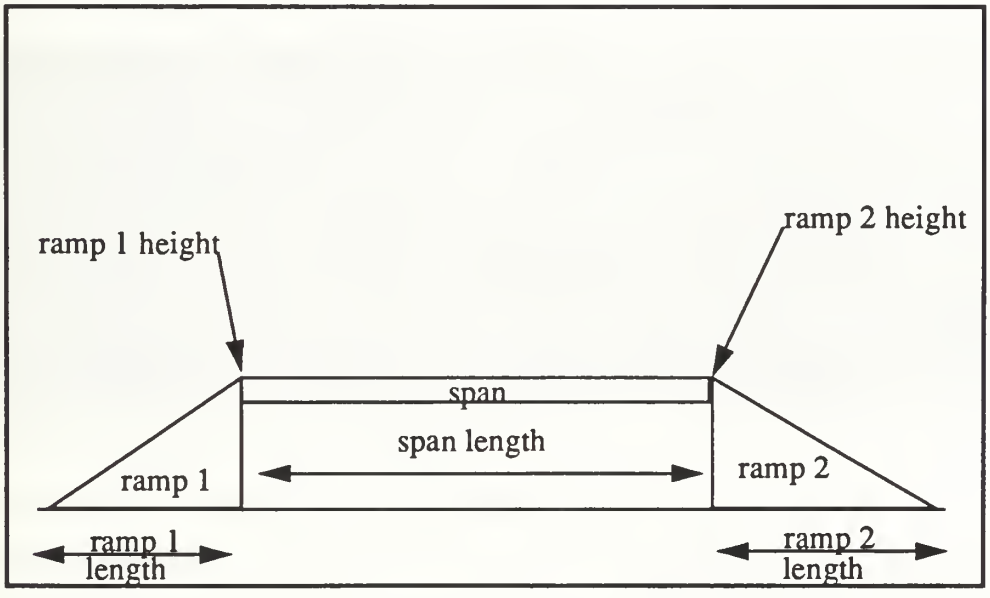



Figure 11: Bridge Cross Section

Bridges in real life can be curved and angled in the direction of travel, but in this work a simpler and straightforward approach was used. Each pair of points that make up the top

of the bridge is set to the same height. This insures that the bridge is level and not bent. This compromise is more than justified, because of the less complicated method to construct a bridge. The bridge can slope, however, from end to end. Bridges cannot be overlapped.

## B. TRAVERSAL

Traversing a bridge is completely different from the manner that earthworks are traversed. Unlike earthworks, a vehicle must be able to cross over and under bridges, and it must be able to fall off of the bridge (Figure 12). One of the major concerns with traversing bridges is not jumping up to the bridge when the vehicle should be going underneath. There must be some method and tag for the vehicle to know that it is on or off a bridge. If a vehicle is on the bridge, it must use the bridge elevations until it is off of the bridge.

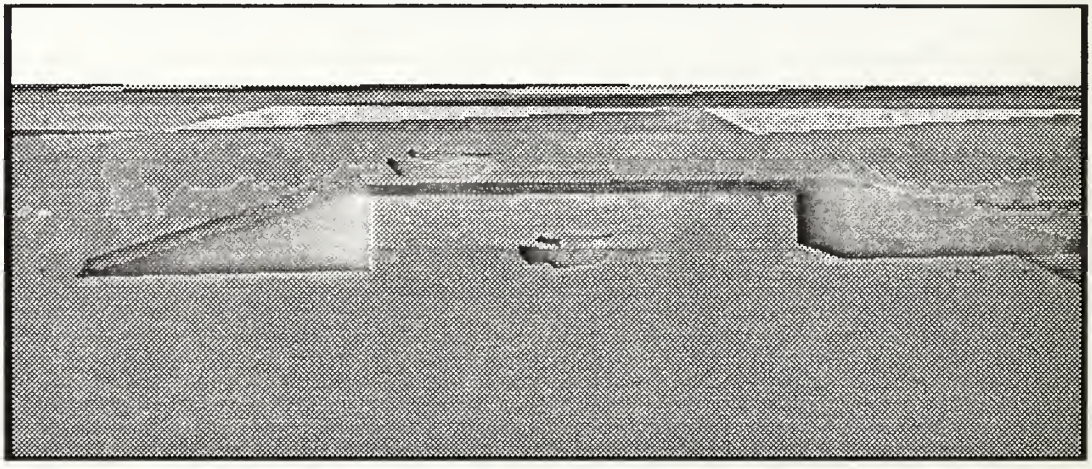


Figure 12: Vehicle Traversal of a Bridge

### 1. Getting on the Bridge

For a vehicle to get on the bridge, it must be properly aligned with the bridge and preceding in the correct direction. For this reason, the vehicle also passes a reference to itself when it makes the call to the terrain. Assuming that the vehicle is not yet on the bridge, the call to the terrain for elevations is made the two point method. The two point

method is always used whenever there are no bridges present in the vehicle area, because it is faster and sufficiently accurate. If a bridge is present the vehicle will switch to the multipoint method. When a bridge is present in the grid square, the terrain class must check to see if the vehicle is within a hot spot. The hot spot is an rectangular area that is centered at the beginning of the bridge and is as wide as the bridge (Figure 13). If the vehicle is not in a hot spot, nothing is done and a zero elevation is returned, but if it is in a hot spot the vehicle direction must be checked. This is the purpose of the reference to the vehicle. Using the reference, the bridge can now query the vehicle for its direction and speed. The direction is used to determine if the vehicle is properly lined up with the bridge, and the speed is needed for the case when the vehicle is backing over the bridge. Once it has been determined that the vehicle is in a hot spot and headed onto the bridge, the vehicle reference is used again to turn the vehicle's bridge tag to true.

If the vehicle is not proper aligned to get on the bridge, but it is in a hot spot the vehicle will still get elevations from the bridge. This allows the vehicle to traverse the end of the bridge in the same manner it would cross a berm.

### 2. Driving Over the Bridge

The vehicle can drive across the ends of the bridge as should be normal, but when the vehicle has driven onto the bridge a new rule holds. If the vehicle's "on bridge" tag is true, the vehicle obtains the elevation from the bridge, otherwise it only receives bridge elevations when in the hot spots. To ensure realism, the vehicle must remain on the bridge to continue receiving the elevations from it. In real life, a vehicle can turn perpendicular to a bridge while in the middle of it, and it can drive off the end of the bridge. The vehicles can drive onto the bridge, stop and turn perpendicular to the direction of the bridge. This allows the players in the simulation to use vehicles to block bridges. Falling off abruptly and driving off normally from the bridge are covered below.

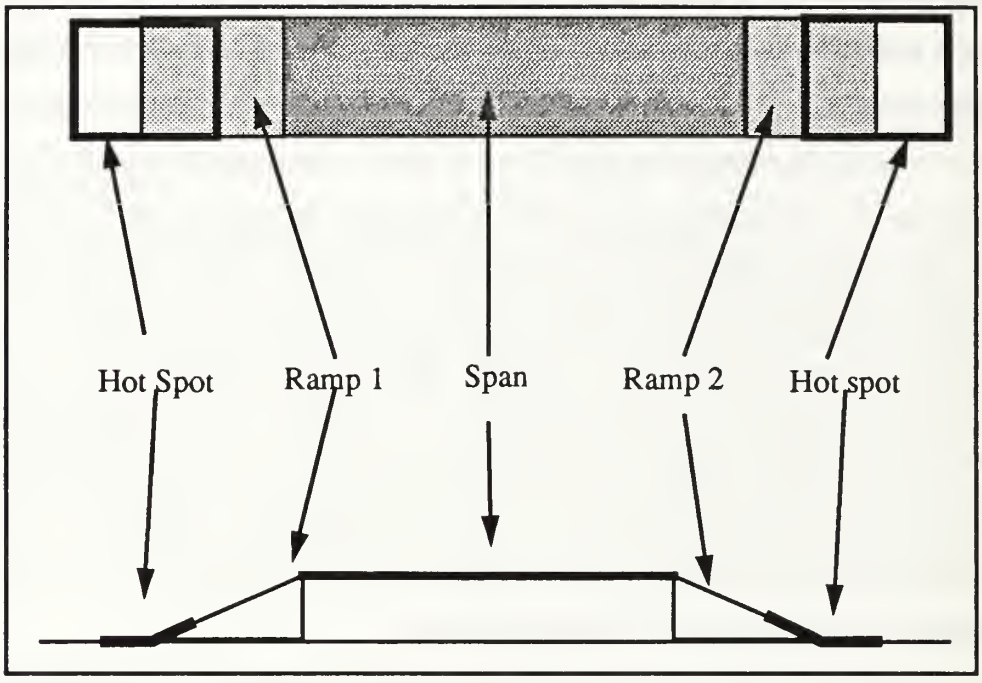



Figure 13: BRIDGE HOT SPOTS

### 3. Driving and Falling Off the Bridge

Once a vehicle is on a bridge, it must be able to get off of the bridge normally or in a more dangerous manner. If a vehicle's "on bridge" tag is set, it remains on the bridge until it reaches a hot spot and is headed off of the bridge. A vehicle must be able to back off the bridge, turn around and drive off the same way it came on (provided the bridge is wide enough), and it should be able to drive off the opposite end of the bridge. The same method that was used to get on the bridge is used to get off the bridge, except that it is in reverse and that only the portion of the hot spot that is beyond the bridge is used. When the vehicle enters the hot spot, the bridge will query the vehicle for its direction and speed, and if they are within limits, the vehicle's "on bridge" tag is set False.

If the vehicle is on a bridge and only one of its tires is off the bridge, it should remain on the bridge. If more that one tire is off of the bridge, the vehicle falls off of the

bridge. The way this is implemented is by letting the vehicle check its pitch and roll. If they are out of bounds, the vehicle turns its "on bridge" tag to false and take its elevations from the ground. The vehicle dies if the pitch or roll was too great.

#### 4. Driving Under the Bridge

The most important part of the vehicle traversal is being able to drive underneath the bridge. If the vehicle's "on bridge" tag is false, the vehicle proceeds under the bridge. This is the real purpose of the "on bridge" tag. This use of the tag prevents the vehicle from jumping up to the bridge and back down to the ground when it is past the bridge.

## C. RUN-TIME MODIFICATION OF BRIDGES

Bridges can be modified during the exercise in the same manner as the earthworks, but bridges can also be destroyed. The simplest method of rendering a destroyed bridge is to not draw the span, but that looks more like an uncompleted bridge. Another method is to remove a portion of the span from the center and place jagged edges on the ends of the open span, but this takes more memory and time. Another method, the method chosen in this work, is to draw the span with the middle of it touching the ground. This method required only four addition points and four additional polygons, and effective represents a destroyed bridge. Vehicles can not traverse the destroyed span.

# V. NETWORKING AND DYNAMIC TERRAIN

Networking of dynamic terrain is an important factor of the realism and functionality of the battlefield simulator. The use of networks allows multiple players spread over a distance, and faster operation by spreading the work load out. This chapter covers issues that relate to networking dynamic terrain, and the Protocol Data Units (PDUs) that are proposed.

## A. COMMUNICATIONS

An important factor of networking is the communication between the nodes or players. What happens in one player's world should also happen in all of the player's worlds. There must be reasonable compromise between sending enough information to the network players to maintain real-time realism and still not overload the channel. The ideal solution is to all amount of information for precise detail of the world, but this is time intensive and overloads the network link. Another difficulty is the difference in machinery. As the simulation is run the features that are so perfectly built on one workstation is possibly buried in the terrain or floating in mid-air on another [LIND 92]. Rather than sending all of the precise measurements or vertices and material properties, a standard containing only the minimum parameters to construct the terrain feature or modify it should be sent. Each of the nodes would then be responsible for managing the information according to their own specifications and hardware limitations.

For one node to communicate with another, there must be a standard of protocol data units that all nodes can understand and manipulate. These PDUs must be consistent among the players in the network and contain only the information needed to manage the construction and modifications of the dynamic features. The standard used for this work is derived from the PDU formats proposed by Lindberg [LIND 92], and based on the Distributed Interactive Simulations (DIS) standard [IST 91].

There are several reasons why one node must communicate with another, but this paper focuses only on the occasions when dynamic features are involved. There are six cases discussed below as follows:

**1. Signing on/off.**

When a node signs on or joins the network during a simulation, it must be informed of the features that exist from other nodes. This can be done by the simulation manager or the node can request the features from the other nodes by itself. The approach proposed by this paper is to let the station manager have all of the nodes transmit their maximum feature ID. Once the node has a maximum ID, it can check it with its internal records and request any missing features. Unless the node is the simulation manager, there is no requirement when the node leaves the network.

**2. Creation of a Feature.**

When a player creates a new dynamic feature it must notify the other players of this new feature.

**3. Modification of a Feature.**

When a player modifies an existing feature it must pass that information to the other players. The PDU for this event is of the same format as the one used to create a feature, but is of a different PDU type identification.

**4. Removing a Feature.**

Whenever a player deletes or removes a feature, the remaining players need to be told to remove the feature from their world as well.

**5. Requesting a Feature.**

Occasionally a player does not receive a feature ID from another player. This can happen for many reasons such as being temporarily off line or communication difficulties. Once the player receives a feature ID that is greater than it expected it must request that the

originating player retransmit the missing events. If the originating player is not on line, the Dynamic Terrain Server is responsible for sending the appropriate information.

### 6. Dynamic Terrain Server.

The dynamic terrain server must keep a record of all the maximum feature ID numbers from each node. When a node requests to join the network, the dynamic terrain server must have the remaining nodes transmit their maximum feature ID to the new node. If the simulation manager must leave the simulation, then it must pass the duty of simulation manager to another node as well as all of its internal list of feature IDs.

## B. DYNAMIC TERRAIN PROTOCOL DATA UNITS (PDU'S)

There are five dynamic feature PDU proposed by this research: Destruct PDU used to remove a feature, Construct PDU and Update PDU are used to create new and modify existing features, Maximum Feature PDU used to notify other nodes of the number of features created by this node, and Playback PDU used to request missing features from another node. The proposed format for these nodes are in the following tables and each of the fields are briefly described.

### 1. Dynamic Feature Construction and Modification PDU.

This PDU is used to send notification of the presence of a new feature in the simulation, and it is also used when there is a change to an existing feature's appearance or status. This PDU is described below and shown in detail in Table 1, "CONSTRUCT / MODIFY PDUs," on page 31.

a. **PDU Header** The standard DIS header.

b. **Node ID** The standard DIS entity format minus the entity ID.

c. **Feature ID** This is a sequential number that is incremented each time a new feature is created. On simulation start-up, it is set to zero, but, on occasion, when a node rejoins the simulation, this number is passed from the dynamic terrain server.

d. **Feature Type** This is the type of feature being referenced. It is a C++ enumerated type and returns a short integer for a value.

e. **Time Stamp** This is the standard DIS time stamp.

f. **Position** This is the starting position of the feature in world coordinates.

g. **Direction** This is the direction of travel from the starting point of the feature, and is only used by the berm and bridge features.

h. **Height** This field contain two floats that represent the height of the feature. The first or primary height is used by berms to set its height, and by bridges to set the height above ground of the first ramp where it joins the suspended portion of the bridge. The secondary height is used only by the bridge feature to set the height of ramp two at the far end of the suspended portion.

i. **Length** This field contains three floats. The primary length is used by all of the features discussed in this paper. It is the radius of the crater, the length of the berm, and the length of the suspended portion of the bridge. The second and third floats are used only by the bridges for the lengths of ramp one and ramp two respectfully.

j. **Material** This is the enumerated material type for the material to be used by the feature for representation.

k. **Status** This field is used by the bridge feature to indicate a destroyed or whole bridge to be displayed.

TABLE 1: CONSTRUCT / MODIFY PDUs

| Field Size | Field Title | Description | |
|---|---|---|---|
| 32 | PDU Header | 8 bits - Char<br>8 bits - Char<br>8 bits - Char<br>8 bits - Char | Protocol Version<br>Exercise identifier<br>PDU type<br>unused |

TABLE 1: CONSTRUCT / MODIFY PDUs (CONTINUED)

| Field Size | Field Title | Description |
|---|---|---|
| 32 | Node ID | 16 bits - Short Integer Site ID<br>16 bits - Short Integer Node ID |
| 16 | Feature ID | 16 bits - Short Integer |
| 16 | Feature Type | 16 bits - Enumerated C++ type |
| 32 | Time Stamp | 32 bits - Integer |
| 96 | Position | 32 bits - float X<br>32 bits - float Y<br>32 bits - float Z |
| 32 | Direction | 32 bits - float |
| 64 | Height | 32 bits - float primary height<br>32 bits - float secondary height |
| 96 | Length | 32 bits - float primary length<br>32 bits - float second length<br>32 bits - float third length |
| 16 | Material | 16 bits - Enumerated C++ type |
| 32 | Status | 32 bits - Integer |

**2. Destruction PDU.**

This PDU is used to remove a feature from the simulation. Description of the PDU follows below and is shown in detail in Table 2, "DESTRUCT PDU," on page 33

a. **PDU Header** See Table 1 for description.

b. **Node ID** See Table 1 for description.

c. **Feature ID** This is the identification number of the feature that is to be removed from the simulation

d. **Feature Type** See Table 1 for description.

e. **Time Stamp** See Table 1 for description.

TABLE 2: DESTRUCT PDU

| Field Size | Field Title | Description |
|---|---|---|
| 32 | PDU Header | 8 bits - Char Protocol Version<br>8 bits - Char Exercise identifier<br>8 bits - Char PDU type<br>8 bits - Char unused |
| 32 | Node ID | 16 bits - Short Integer Site ID<br>16 bits - Short Integer Node ID |
| 16 | Feature ID | 16 bits - Short Integer |
| 16 | Feature Type | 16 bits - Enumerated C++ type |
| 32 | Time Stamp | 32 bits - Integer |

### 3. Maximum Feature ID PDU.

This PDU is used by a node to notify other players of its total number of features that have been created. The PDU is shown in detail in Table 3, "MAXIMUM FEATURES PDU," on page 34.

a. **PDU Header** See Table 1 for description.

b. **Node ID** See Table 1 for description.

c. **Maximum Feature ID** The number of the last feature created by this node.

d. **Time Stamp** See Table 1 for description.

TABLE 3: MAXIMUM FEATURES PDU

| Field Size | Field Title | Description |
|---|---|---|
| 32 | PDU Header | 8 bits - Char Protocol Version<br>8 bits - Char Exercise identifier<br>8 bits - Char PDU type<br>8 bits - Char unused |
| 32 | Node ID | 16 bits - Short Integer Site ID<br>16 bits - Short Integer Node ID |
| 16 | Maximum Feature ID | 16 bits - Short Integer Total # of Features |
| 32 | Time Stamp | 32 bits - Integer |

**4. Playback PDU.**

This PDU is used to request a retransmission of features from another node. This might be used when a node has been off line temporarily or if for some reason it missed the transmission from the sending node. The initiating node sends this when it receives a feature ID from a node that does not match its internal value for the next node. The PDU is shown in detail in Table 4, "PLAYBACK PDU," on page 35.

a. **PDU Header** See Table 1 for description.

b **Requesting Node** This is the node that is requesting the retransmission.

c. **Transmitting Node ID** This is the node that is to retransmit the missing features.

d. **Start Feature ID** This is the number of the feature to begin transmission with.

e. **End Feature ID** This is the last known maximum feature ID and is the stopping point of the transmission.

f. **Time Stamp** See Table 1 for description.

TABLE 4: PLAYBACK PDU

| Field Size | Field Title | Description |
|---|---|---|
| 32 | PDU Header | 8 bits - Char Protocol Version<br>8 bits - Char Exercise identifier<br>8 bits - Char PDU type<br>8 bits - Char unused |
| 32 | Requesting Node ID | 16 bits - Short Integer Site ID<br>16 bits - Short Integer Node ID |
| 32 | Playback Node ID | 16 bits - Short Integer Site ID<br>16 bits - Short Integer Node ID |
| 16 | Start Feature ID | 16 bits - Short Integer |
| 16 | End Feature ID | 16 bits - Short Integer |
| 32 | Time Stamp | 32 bits - integer |

## C. SIMULATION RECOVERY

In networking, one must consider what happens when the dynamic terrain server fails or when the entire system fails. Another consideration is the temporary suspension of the simulation until a later date, to be resumed where it was stopped.

As far as part or all of the system failing, the standard switching of roles would occur as necessary. The dynamic terrain server would take over any duties of the missing player until such time as the player returns. If the server itself is the part that failed, then a new server would be dedicated.

When a failure does occur, the dynamic terrain manager saves the terrain to a file to be reused later when the system is on line. This is also done on normal shutdown so the

simulation can be restarted from the last status of the terrain. The implementation of dynamic terrain in this research did not include networking, but saving the terrain to a file has been implemented. The program can be started from default or by reading from a file.

# VI. CONCLUSIONS AND FUTURE WORK

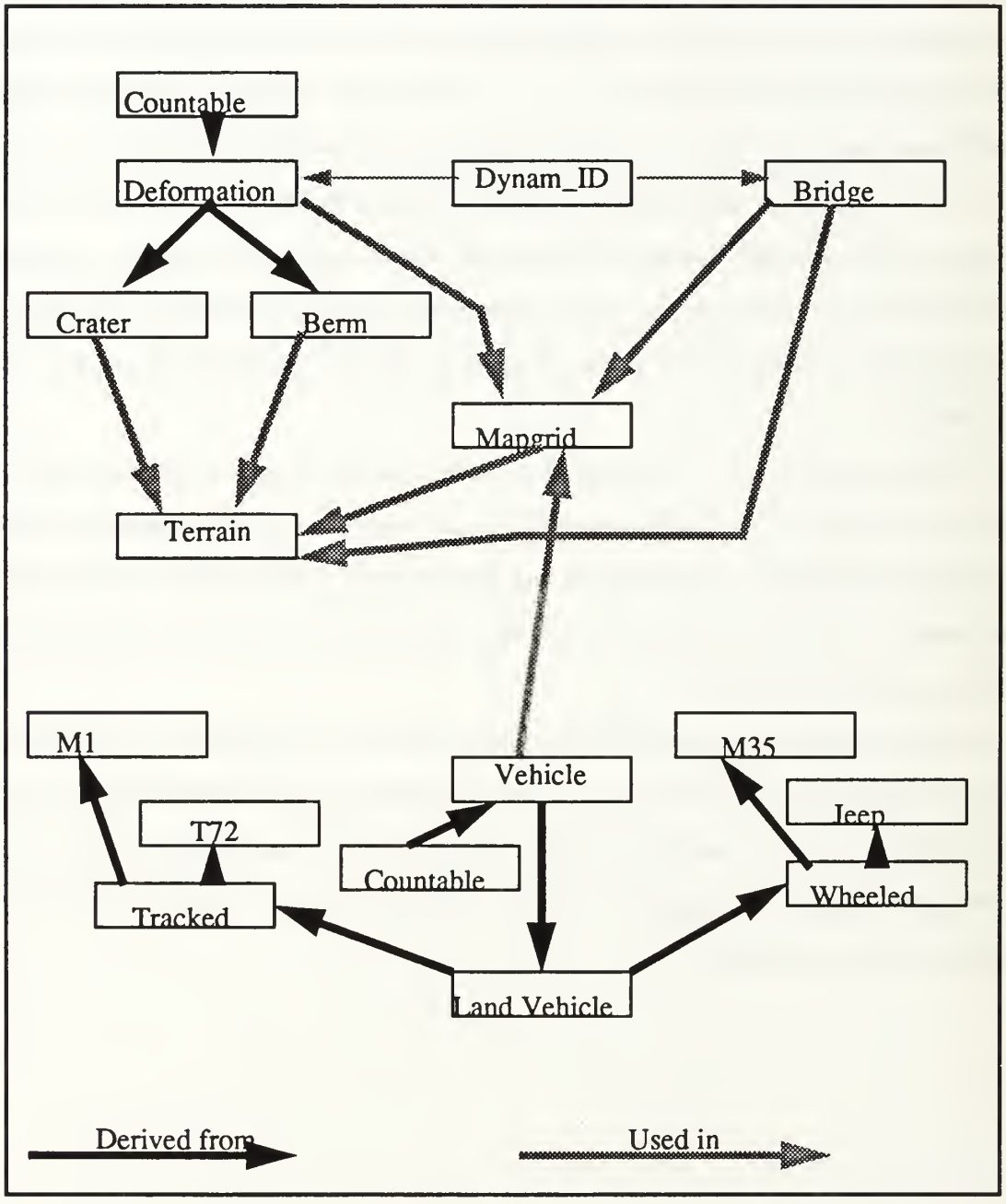
This work has implemented two types of dynamic terrain (berm and craters), bridges, and vehicle traversal of same. This work has drastically improved the realism of NPSNET through the addition of the dynamic terrain. The dynamic terrain was implemented using C++ and object oriented programming. This also provides a basis for the transformation of the current version of NPSNET from C to C++. The terrain class manages all of the objects that are attached to it by calling methods of the objects. The object oriented capabilities of C++ classes were beneficial in allowing the class objects to take care of any requirements it has to complete a task. Vehicles traversing the terrain only have to ask the terrain class for the elevations of its orientation points. The terrain takes care of asking the bridges and deformations for the elevations, and the features themselves perform all the computations as necessary.

The benefits of using C++ and an object oriented design make it impracticable not to redesign NPSNET to C++ and incorporate the new terrain and dynamic feature developed and implemented in this work. Vehicles can drive over and under bridges, and over berms and craters in a realistic manner. Also implemented in this work was the physically based model vehicles [PARK 92].

Future work on dynamic terrain includes implementing ditches, and real rivers and lakes. Continued work on improving the object oriented design of NPSNET is imperative. Networking the dynamic terrain and implementing the use of multiprocessor to speed the system up would also be extremely helpful. Levels of resolution for the dynamic terrain should also be implemented.

## APPENDIX A CLASS HEIRARCHY

This appendix is the dependency and class hierarchy diagram for the implementation of dynamic terrain.



## APPENDIX B CLASS DESCRIPTIONS

Many of the data members used in this work were provided through the C++ class libraries NPSCL [WILS 91] and NPSGDL [WILS 92].

# CLASS DYNAMIC_ID

This class is the identification of the bridges and deformations. It is designed with networking in mind, so it includes space for site and host ID as well as the sequential number of the feature created. Using a common identification system reduces the confusion and complexity of the system. The static data members and member functions are always available through the scope operator.

### Data members

| | |
|---|---|
| static int last_id; | //The number of the next available ID |
| static int total; | //The total number of features in the exercise |
| int site; | //The site identification |
| int host; | //The host identification |
| int ID; | //The identification of the feature |

### Member Functions

**dynamic_id**();

The default constructor initailizes all parts to zero.

**dynamic_id**(int S, int H, int id);

This constructor is used when reading an ID from a file. It does not increment the count of ID numbers used.

**dynamic_id**(dynamic_id& d_id);

The copy constructor.

static void **resume_count**(int cnt);

This function is used when starting the program from a file that has features.

void **assign_id**();

This function is used to increment the counter and assign it to the ID of the feature.

static int **last_count**();

This function returns the value of the next available ID number.

static int **num_of_features**();

This function returns the numbers of features currently in the exercise. This is not the same as the ID counter.

int **get_site**();

This function returns the site number of the feature.

int **get_host**();

This function returns the host number of the feature.

int **get_ident**();

This function returns the ID number of the feature.

bool **operator==**(dynamic_id& d_id);

This function checks all of the fields of the ID for equality, and returns true if they are equal.

bool **operator<**(dynamic_id& d_id);

This function returns true if d_id is greater than this id.

bool **operator>**(dynamic_id& d_id);

This function returns true if d_id is less than this id.

void **write_id**(ostream& file);

This function writes the ID to a file.

static dynamic_id& **read_id**(istream& file);

This function reads the file for the identification of a feature.

**~dynamic_id**();

The class destructor

## CLASS DEFORMATION

Class deformation is derived from the class countable RRRRR so that use of reference counting pointers and shared memory is available. The deformation class is the data and functions that are common to all types of dynamic terrain. There are some functions listed below that are virtual. These functions are repeated by name in all of the derived classes and must be listed in any future derived class of deformation. The allow the user to use a pointer to the base class and make a virtual function call without worrying about what type of object is actually pointed to. For example, the deformation could be a crater or a berm, but in either case the same call to display will cause the feature at the end of the pointer to display itself. The deformation class must know that the terrain does exist, but does not need to all of the details, so there is a bit of double dependency.

**Data members**

| | |
|---|---|
| terrain *ter_ptr; | //pointer to the terrain to use |
| vertex posit; | //position of deformation |
| dynamic_id def_id; | //ident # of deformation |
| soil_types soil_type; | //type of soil |
| Array(mgptr) point_me; | //which grid points should point to this def |
| bool displayed;/ | /has this def been displayed yet |

**Member Functions**

**deformation**();

This is the default constructor. All data members initialized to zero.

**deformation**(terrain *ptr, soil_types soil);

This constructor identifies the underlying terrain and the soil type of the terrain.

**deformation**(terrain *ptr, vertex v);

This constructor takes a pointer to the underlying terrain and a position of placement. The soil type is taken from the underlying terrain.

**deformation**(terrain *ptr, vertex v,soil_types soil);

This constructor takes the terrain pointer, position of placement, and the material type for a deformation object.

**deformation**(const deformation& def);

This construct builds a defomation object from another deformation object.

virtual ~**deformation**();

This is the class destructor.

vertex **get_posit**();

This function returns the placement position of a deformation to the requester.

void **set_posit**(vertex v);

This function allows the placement position of a deformation to be changed.

dynamic_id& **who_you**();

This function returns the identification number of a deformation.

void **set_soil**(soil_types s);

This function allows a user to change the soil material type of a deformation.

soil_types **get_soil**();

This function returns the soil property type for a deformation.

terrain ***get_terrain_ptr**();

This function returns a pointer to the underlying terrain being used by a deformation.

void **set_terrain_ptr**(terrain *t);

This function is used to initialize or change the underlying terrain to which the

deformation must attach itself to.

void **display_off**();

This function is used to turn the displayed flag off. The flag can only be turned on by the deformation during display or writing to a file.

virtual void **calculate**();

This function is used by the derived classes to compute the 3D position for the points of the structure.

virtual void **pre_destruct**();

This function is used when deleting a deformation to ensure that all grid squares that have a reference pointer to this deformation from their linked list.

virtual void **display**();

This function tells the class derived from deformation to display itself.

virtual bool **made_contact**(float xpos, float zpos);

This function is used to determine if a point is in bounds of a deformation, it does not consider height only ground position.

virtual float **how_high**(float xpos, float zpos);

This is the function that returns the elevation of a point on a deformation to the requesting user. If the point is not on a deformation, the deformation returns a zero.

virtual void **store_2**(ostream& file);

This function is used to write the deformation's information to a file.

static void **read_from**(istream& file, terrain *Tptr);

This is the function is used to read from a file the information needed to construct a deformation. It then calls a special private constructor to build the deformation and ensure that the identification numbers remain non-repetitive.

## CLASS CRATERS

The crater class is derived form the deformation class and contains all of the additional information needed to construct and used craters. The crater objects can be accessed directly or through the deformation pointers and virtual functions. In order to access the crater objects directly, the deformation pointer must be cast to a crater pointer.

**Data members**

float radius; //radius of the crater

vertex sector[GRIDS][RINGS]; //points of the crater

vertex normals[GRIDS][RINGS]; //normals of the crater

There are sixteen GRIDS and four RINGS in a crater.

**Member Functions**

**crater**();

This is the default constructor for craters. All data member initialized to zero.

**crater**(terrain *Tptr, vertex pos, float diam, soil_types soil);

This is a crater constructor that takes a pointer to the underlying terrain, a placement vertex, a diameter for the crater, and a soil property type.

**crater**(terrain *Tptr, vertex pos, float diam);

This is a crater constructor that takes only the terrain pointer, the placement point and the diameter. The soil type is taken from the terrain.

**crater**(const crater& crat);

This copy constructor builds a crater object from another crater object.

**~crater**();

This is the crater destructor

float **get_radius**();

This function returns the radius of the crater.

void **set_radius**(float diam);

This function changes or initializes the crater radius.

float **pull_xpos**(int a, int b);

This function returns the x value of a crater point at index a,b.

float **pull_zpos**(int a, int b);

This function returns the z value of a crater point at index a,b.

float **pull_yheight**(int a, int b);

This function returns the y or height value of a crater point at index a,b.

vertex **pull_all**(int a, int b);}

This function returns the vertex values of crater point at index a,b.

void **set_xpos**(int a, int b, float pos);

This function sets the x value of the point at index a,b. This useful to create special looking craters for erosion and breaching.

void **set_zpos**(int a, int b, float pos);

This function sets the z value of the point at index a,b. This useful to create special looking craters for erosion and breaching.

void **set_yheight**(int a, int b, float height);

This function sets the y or height value of the point at index a,b. This useful to create special looking craters for erosion and breaching.

void **set_all**(int a, int b, float xpos, float ypos, float zpos);

This function sets the vertex value of the point at index a,b. This useful to create special looking craters for erosion and breaching.

virtual void **calculate**();

This function is called to compute the points of the crater and the height of the crater. It also calls a private function to compute the normals for the crater points. This function will also take care of placing the crater on the terrain and being put on the linked list of reference counting pointers for each grid square it has touched.

virtual void **pre_destruct**();

This function is used when removing a crater object from the world. It ensures that all of the mapgrid squares remove this crater from their list of deformations.

virtual void **display**();

This functions causes the crater to display itself.

virtual bool **made_contact**(float xpos, float zpos);

If a point (on the ground) is within the bounds of the crater, then this function returns true.

virtual float **how_high**(float xpos, float zpos);

This is the function that returns the elevation of the crater at a point. A zero return means that the point was outside the bounds.

virtual void **store_2**(ostream& file);

This function writes the important crater information to a file so that the crater can be reused later.

static void **read_crater**(istream& file, terrain *Tptr);

This function reads the crater information from a file and is passed the pointer of the terrain to use. Then a private constructor is called to build the crater and ensure that a valid ID is used.

## BERMS

Berms are basically mounds of earth that have been pushed up for one reason or the other. These berms mold to the underlying terrain and can have any type of material property. Berms use the same enumerated type for object material as the craters, but berms can be given a different material type, for example, the user might want a concrete barrier on ground that is red clay. The berms are derived from the deformation class. Berms are built from the placement point and run the distance of the length and in the direction provided.

**Data members**

```
float height;                  //height above the ground
float width;                   //width at the base
float length;                  //The length of a berm
float direction;               //the theta of the berm - 0 is +x direction
vertex bermgrid[XLINES][ZLINES]; //the vertices of the berm
vertex normals[XLINES][ZLINES];  //the normals of the berm
```

There are four ZLINES in a berm and eleven XLINES.

**Member Functions**

**berm**();

This is the default berm consructor. All data members are initialized to default values.

**berm**(terrain *Tptr, vertex pos, float hgt, float wdt,

float lgt, int dir, soil_types soil);

This Constructor takes all of the parameters to build and place a berm: a pointer to the underlying terrain, the placement position, height above ground, length, width, the direction from the placement point, and the material type.

**berm**(terrain *Tptr, vertex pos, int dir);

This constructor takes a pointer to the underlying terrain, a placement position, and a direction, and then uses the defaults for the remainder of the data members.

**berm**(berm& B);

This is the copy constructor that creates a berm from another berm.

~**berm**();

This is the class destructor.

float **get_height**();}

This function returns the height of the berm.

void **set_height**(float hgt);

This function is used to change the height of the berm.

float **get_width**();

This function returns the width of the berm.

void **set_width**(float wdt);

This function is used to change the width of the berm.

float **get_length**();

This function returns the length of the berm.

void **set_length**(float lgt);

This function is used to set or change the length of the berm.

float **get_direction**();

This function returns the direction of the berm. A direction of zero is pointing to the positive x axis. A positive increase of the direction moves the berm in a counter-clockwise manner.

void **set_direction**(float dir);

This function is used to set the direction of the berm.

static void **read_file_berm**(istream& file, terrain *Tptr);

This function is used to read the information needed to build a berm from a file. This function is always available to the user because it is static. It is called using the Scope operator, (ex. berm::read_file_berm(file, terrain). Once the file has been read, the special constructor is called. The special constructor is private and unavailable to the user. Its purpose is to maintain identification interegrity

virtual void **calculate**();

This function is used to calculate the points of the berm. This must be called whenever a dimension of the berm changes. Once the points of the berm have been calculated the function will call the private function to compute the normals of the points.

virtual void **pre_destruct**();

This function is used when removing or deleting a berm from the terrain. It ensures that all of the grid squares remove this berm from their linked list of deformation.

virtual void **display**();

This function displays the berm. The berm will only be displayed once, unless the user resets the display through the deformation display_off() function.

virtual float **how_high**(float xpos, float zpos);

This function returns the elevation of a point anywhere on the berm.

virtual bool **made_contact**(float xpos, float zpos);

This function is a quasi bounding box, but only for an x and z position. If the point past in is within the berm, this function will return true.

virtual void **store_2**(ostream& file);

This function writes the berm to a file.

## BRIDGES

The bridge class is basically the same as the deformation and berm classes, but is different because of the differences required for vehicle traversals. The bridge class must also know about the existence of terrain class. The bridge is made up of three sections: two ramps and one span. There are two heights for the bridge, one for each ramp end, where it joins the span. The bridge has only one width. Each section of the bridge has a length. The bridge is designed so the placement is at the joint of the span and ramp one. This allows the span to be set to cover a specified distance, and the have the on/off ramps extend from the ends of the span. For a vehicle to get on the bridge, it must be aligned properly, in a hot spot, and headed on to the bridge. The hot spot is a rectangle area with a length equal to the width of the bridge, and centered on the ramp where it joins the terrain. The vehicles must pass a reference to themselves to the bridge, so that the bridge can determine the vehicle's alignment and direction in relation to that of the bridge. The bridge class is designed so that other classes of bridges can be designed and used, just like the berms and craters are to deformations.

**Data members**

```
terrain *ter_ptr;          //pointer to the terrain to use
vertex posit;              //position of bridge
int bridge_id;             //ident # of bridge
float groundatcenter;      //elev of grnd at mid of bridge
float width;               //width of the bridge
float R1_length;           //length of ramp one
float R2_length;           //length of ramp two
float flat_length;         //length of flat
float flat_hgt_R1;         //hgt of bridge at end of ramp 1
```

```
float flat_hgt_R2;              //hgt of bridge at end of ramp 2
bool destroyed;                 //Is bridge destroyed
bridge_mattypes bridge_mat;  //material for the bridge
Array(mgtpr) point_this_bridge;//which grids point to this bridge
vertex bridge_pts[4][2];        //points for the bridge
vertex bridge_btm[2][2];        //two point for the bottom
```

**Member Functions**

**bridge**();

This is the default construct. The data members are set to prearranged default values.

**bridge**(terrain *ptr,vertex pos);

This constructor takes only the pointer to the underlying terrain and a placement point, the rest of the data members will use defaults.

**bridge**(terrain *ptr, vertex pos, bridge_mattype bmat);

This constructor takes only the pointer to the underlying terrain, a placement, and a material to use for construction.

**bridge**(terrain *ptr, vertex pos, bridge_mattype bmat, float dir);

This constructor takes a terrain pointer, a placement position, a material to use, and a direction to run.

**bridge**(terrain *ptr, vertex pos, bridge_mattype bmat, float dir,
float width, float height, float length);

This constructor takes a terrain pointer, a placement point, a material type, a direction to run, a width, a height to be used on both ends of the span, and an overall length of the entire bridge.

**bridge**(terrain *ptr, vertex pos, bridge_mattype bmat, float dir,
float width, float hgt_r1, float hgt_r2, float len_r1,
float len_top, float len_r2);

This constructor takes all of the parameters to build and place the bridge.

**bridge**(const bridge &B);

The copy constructor.

virtual ~**bridge**()

The destructor.

vertex **get_posit**();}

This function return the placement point of the bridge.

void **set_posit**(vertex v);

This function sets or changes the placement point.

terrain ***get_terrain_ptr**();

This function returns a pointer to the underlying terrain being used by the bridge.

void **set_terrain_ptr**( terrain *t);

This function is used to set or change the underlying terrain being used by the bridge.

void **set_width**(float wdt);

This function is used to set or change the width of the bridge.

float **get_width**();

This function returns the width of the bridge.

void **set_R1_length**(float l);

This function is used to change the length of the ramp closest to the placement point, called ramp one.

void **set_R2_length**(float l);

This function is used to change the length of the ramp furthermost from the placement point, called ramp two.

void **set_flat_length**(float l);}

This function is used to change the length of the span.

float **get_R1_length**();

This function returns the length of the ramp closest to the placement point,

ramp one.

float **get_R2_length**();}

This function returns the length of the ramp furthermost from the placement point, ramp two.

float **get_flat_length**();}

This function returns the length of the span.

void **set_bridge_destroyed**();

This function is used to set the bridge destroyed flag to true. When the bridge is destroyed the middle of the span will touch the ground.

void **restore_bridge**();

This function is used to set the bridge destroyed flag to false. Used to repair a destroyed bridge.

bool **is_destroyed**();

This function returns true if the bridge has been destroyed.

void **set_bridge_mat**(bridge_mattype t);

This is used to changed the material used to build the bridge. This is from list of available material from the enumerated list of soil types.

bridge_mattype **get_bridge_mat**();

This functions returns the type of material being used by the bridge.

virtual void **calculate**();

This function is used to build or compute the points of the bridge. It calls the private function to calculate the normals of the polygons of the bridge.

virtual float **bridge_elev**(float xpos, float zpos);

This function is similar to the how_high0 functions of the deformation derived classes, it returns the elevation of a point on a bridge IF the vehicle is either in a hot spot or is on the bridge.

virtual void **display**();

This function displays the bridge according to its status, normal or destroyed.

## MAPGRID

This is the class of mapgrid points. It is the C++ version the struct mapgridstruct, used in the C version of NPSNET. The mapgrid point is information control center for one grid square. It contains linked list of reference counting pointer to deformations, bridges, and vehicles. The use of the reference counting allows many mapgrid points to point to the same object, specifically the deformations and bridges, while only having one copy in memory. Many of the member functions of this class only call the member functions of deformation, bridges, and vehicles that do the same function. The purpose of this is to provide the user access to the functions of the objects without having to know the type of object. This class also provides the necessary list management functions needed to add, access, remove items from the lists. The mapgrid point class is called mgridpnt, and correspond to a corner of a grid square.

### Data members

```
float elevation;                    //elevation of terrain
vertex norm;                        //The normal of this node
soil_types upper_soil;              //soil type for the upper triangle
soil_types lower_soil;              //soil type for the lower triangle
List(Refptr_deformation) deform; //Linked list of defs attached
List(Refptr_bridge) brij;           //Linked list of bridges attached
List(Refptr_Vehicle) veh;           //Linked list of vehicles
```

### Member Functions

**mgridpnt**();

This is the default construction for a mapgrid point.

**mgridpnt**(float elev);

This constructor initializes the elevation value of the mapgrid point as it is constructed.

**mgridpnt**(mgridpnt &MG);

This is the copy constructor.

**~mgridpnt**();

This is the destructor.

float **elev**();

This function returns the elevation of the terrain at this mapgrid point

void **elev**(float E);

This functions sets or changes the elevation at this point.

vertex **normal**();

This function returns the value of the normal at this mapgrid point.

void **normal**(vertex &N);

This function is used to set the normal at this mapgrid point.

soil_types **soil_upper**();

This function returns the soil type of the upper triangle of the grid square.

soil_types soil_lower();

This function returns the soil type of the lower triangle of the grid square.

void **set_upper_soil**(soil_types s);

This function is used to set or change the soil type of the upper triangle of the grid squares.

void **set_lower_soil**(soil_types s);

This function is used to set or change the soil type of the lower triangle of the grid square.

void **read_from**(istream& file);

This function reads a mapgrid point from a file. This function does not read in the items of the linked list. That is done separately.

void **store_on**(ostream& file);

This function saves the information, except for the linked list, to a file.

void **store_features**(ostream& file);

This is the function used to save the deformations and bridges in the linked list of this mapgrid point to a file.

void **features_off**();

This function goes through the bridge and deformation linked list turning off feature's display flag.

bool **any_defs**();

This function checks to see if there are any deformations in this point's deformation list

bool **def_valid**();

This function is used when walking through the linked list of deformations, the current pointer, and returns true if there is a deformation at the end of the pointer.

void **first_def**();

This function moves the deformation list iterator to the beginning of the list.

void **last_def**();

This function moves the deformation list iterator to the end of the list of deformations.

void **next_def**();}

This function increments the current pointer to the next deformation in the list.

Refptr_deformation& **curr_def**();}

This function returns a reference counting pointer to the deformation being pointed to by the current pointer.

void **add_def**(Refptr_deformation& dp);

This function is used to add a deformation to the end of the linked list of deformations.

Refptr_deformation& **locate_def**(dynamic_id& ID);

This function returns a reference counting pointer to a deformation, after

locating it by its identification number.

Refptr_deformation& **locate_def**(float xpos, float zpos);

This function returns a reference counting pointer to a deformation, after locating it by a position. It will return the first deformation that includes this point.

void **remove_def**(Refptr_deformation& dp);

This function is used to remove a deformation from this mapgrid point's linked list of deformations, that is pointed to by the pointer dp.

void **remove_alldefs**();

This function empties the deformation linked list of all deformations.

void **display_defs**();

This function displays all of deformation in the linked list

void **recalc_defs**();

This function is used to have all of the deformations in the linked list recalculate themselves.

float **height**(float xpos, float zpos);

This function checks the deformations in the linked list for a deformation at the given point, and returns the elevation at the point. The highest elevation is returned.

bool **def_hit**(float xpos, float zpos);

This function returns true if the point given is within a deformation.

soil_types **def_soil**(float xpos, float zpos);

This function returns the material (soil) type of the deformation at the given point, if a deformation is there.

bool **any_bridge**();

This function returns true if there is a bridge in the bridge linked list at this mapgrid point.

bool **bridge_valid**();}

This function returns true if the bridge list's current pointer is pointing to a bridge.

void **first_bridge**();}

This function moves the bridge list's current pointer to the first bridge in the list.

void **last_bridge**();

This function moves the bridge list's current pointer to the last bridge in the list.

void **next_bridge**();

This function increments the bridge list's current pointer to the next bridge in the list.

Refptr_bridge& **curr_bridge**();

This function returns a reference counting bridge pointer to the bridge being pointed to by the current pointer.

void **add_bridge**(Refptr_bridge& bp);

This function add the bridge pointed to by bp to the end of the bridge linked list.

Refptr_bridge& **locate_bridge**(dynamic_id& ID);

This function returns a reference to a bridge, after locating it in the bridge list of this mapgrid point by the identification number.

Refptr_bridge& **locate_bridge**(float xpos, float zpos);

This function returns a reference to a bridge, after locating it in the bridge list of this mapgrid point by the point passed in. It returns the first bridge that has this point.

void **remove_bridge**(Refptr_bridge& bp);

This function removes the bridge pointed to by bp from this mapgrid point's linked list.

void **remove_allbridge**();

This function empties the bridge linked list for this mapgrid point.

void **display_bridge**();

This function is used to display all of the bridges in this mapgrid point's list.

void **recalc_bridge**();

This function is used to have all of the bridges in the list recalculate themselves.

float **bridge_height**(float xpos, float zpos, Refptr_Vehicle& Veh);

This function returns the elevation of bridge at point (xpos,zpos). The vehicle reference is needed for alignment determination and setting the vehicle on bridge tag on/off as necessary.

bool **bridge_hit**(float xpos, float zpos);

This function returns true is point (xpos,zpos) is within the bound of a bridge in this mapgrid point's linked list of bridges.

soil_types **bridge_soil**(float xpos, float zpos);

This function returns the bridge material type of the bridged located at (xpos,zpos) and in this mapgrid point's list of bridges.

bool **any_veh**();

This function returns true if there are vehicles in the list at this mapgrid point.

bool **veh_valid**();

This function returns true if the vehicle's list current pointer is pointing to a vehicle, otherwise it returns null.

void **next_veh**();

This function increments the vehicle list's current pointer to the next vehicle in the list.

void **first_veh**();

This function move the vehicle list's current pointer to the first vehicle in the list.

void **last_veh**();}

This function moves the vehicle list's current pointer to the last vehicle in the list.

Refptr_Vehicle& **cur_veh**();

This function returns a reference counting pointer to vehicle that is pointed to by the vehicle list's current pointer.

void **add_veh**(Refptr_Vehicle &v);

This function adds the vehicle pointed to by v to the tail of this mapgrid point's linked list of vehicles.

void **remove_veh**(Refptr_Vehicle &v);

This function removes the vehicle pointed to by v from the list of vehicles at this mapgrid point.

void **remove_allveh**();

This function empties this mapgrid point's linked list of vehicles.

void **display_veh**();

This function displays all of the vehicles in the vehicle list at this mapgrid point.

## TERRAIN

The class terrain is the manager for the exercise. It knows about all of the deformations, bridges, and vehicles that are in the exercise. It is responsible for the interaction between objects in the exercise. When a vehicle needs an elevation of the terrain, in need only query the terrain class. The terrain class then queries the bridges and deformations for the proper elevation to use. The terrain is an nXn matrix of mapgrid points.

### Data members

```
int number_of_grid;          //# of gridlines in the terrain
int size_of_grid;            //distance btwn each gridline
static const int bitgridsize; //sqr area cvrd by bits of the Bitgrid
mgridpnt **ground;           //pointer to the array of grid nodes
Bitgrid *def_present;        //used to indicate a feature.in the area
```

### Member Functions

**terrain**();

The default constructor.

**terrain**(int num, int sz);

This constructor builds a terrain of numXnum, with a distance of sz between mapgrid points.

void **init_terrain**(int num, int size);

This function is used to build up a terrain on an existing terrain pointer. It will build up a terrain of numXnum, with size distance between the mapgrid points.

**~terrain**() {}

The terrain destructor

void **read_file**(const string&);

This function is used with the menu item "Terrain - File". It reads an elevation file that is formatted for the C version of NPSNET.

void **read_terrainfile**(const string& filename);

This function reads a terrain file that has been saved by this class. It first builds the terrain from the number of points and size specified at the beginning of the file. Then it reads in the mapgrid points, and after that it reads in any features that are listed.

void **save_terrain**(const string& filename);

This function saves the current state of the terrain to file.

void **set_flat**(float elev);

This function is used with the menu item "Terrain- Flat". It sets the elevation of all mapgrid points to the value of elev.

int **gridsize**();

This function returns the distance between the mapgrid points for this terrain.

int **gridnum**();}

This function returns the number of gridpoints in this terrain.

int **maxsize**();

This function returns the actual size of the terrain.

void **set_elev**(int a, int b, float e);

This function is used to set the elevation at mapgrid point [a][b] to the value of e.

float **get_elev**(int a, int b);

This function is used to return the elevation at mapgrid point [a][b].

void **calc_norm**();

This function is used to calculate the normals of the terrain grid.

void **display**();

This function is used to display the entire terrain grid and all of the objects in it. Not recommend to be used unless a very small terrain and limited objects and features are used.

void **display**(float xpos, float zpos);

This function is used to display only the five gridsquares on each side of the point (xpos,zpos), and all of the objects and features in these gridsquares.

void **feature_display_off**();

This function is used to turn off all of the display flags of the deformations and bridges. Primarily used at the end of the display loop or in writing the features to file.

void **feature_display_off**(float xpos, float zpos);

This function is used to turn off the display flags of the deformations and bridges that are within the moving display box around point (xpos,zpos).

bool **inbounds**(float xpos, float zpos);

This function returns true if the point (xpos,zpos) is on the terrain field, and not out of bounds.

bool **x_inbounds**(float xpos);

This function returns true if xpos is on the terrain and in bounds.

bool **z_inbounds**(float zpos);

This function returns true if zpos is on the terrain and in bounds.

float **nodef_elev**(float xpos, float zpos);

This function is used to find the elevation of the terrain at point (xpos,zpos) without checking for bridges or deformations. Used primarily in placing bridges and deformations.

float **groundelev**(float xpos, float zpos, Refptr_Vehicle& veh);

This function is used to get the elevation of the terrain at point (xpos,zpos). The vehicle reference is used by the bridges for alignment checking and bridge tag setting.

mgridpnt& **gridpoint**(int i, int j);

This function returns a reference to the mapgrid point [a][b].

mgridpnt* **touched_grid**(float xpos, float zpos);

This function returns a pointer to mapgrid that controls the grid square that contains the point (xpos,zpos). Used by the bridge, crater, and berm classes when constructing.

soil_types **get_soil**(float xpos, float zpos);

This function returns the soil material of the terrain, deformation or bridge at point (xpos,zpos).

float **stat_soil_frict**(float xpos, float zpos, int veh_type);

This function returns the static coefficient of friction between non-moving vehicles and the terrain soil type at point (xpos,zpos).

float **dyn_soil_frict**(float xpos, float zpos, int veh_type);

This function returns the dynamic coefficient of friction between non-moving vehicles and the terrain soil type at point (xpos,zpos).

void **new_def**(Refptr_deformation& dp);

This function adds the deformation pointed to by dp to the terrain.

Refptr_deformation& **find_def**(dynamic_id& ID);

This function returns a reference counting pointer to a deformation after locating it by identification number.

Refptr_deformation& **find_def**(float xpos, float zpos);

This function returns a reference counting pointer to a deformation after locating it at point (xpos, zpos).

void **eliminate_def**(Refptr_deformation& dp);

This function removes the deformation pointed to by dp from the terrain.

void **remove_def**(int i, int j);

This function removes all of the deformations at mapgrid point [i][j].

void **remove_alldefs**();

This function removes all of the deformations from the terrain. Used to reset.

void **def_recalc**();

This functions is used to recalculate all of the deformations on the terrain.

void **def_recalc**(int i, int j);

This function is used to recalculate all of the deformations at mapgrid point [i][j].

bool **def_contact**(float xpos, float zpos);

This function returns true if there is a deformation at point (xpos,zpos)

void **new_bridge**(Refptr_bridge& bp);

This function is used to add the bridge pointed to by bp to the terrain.

Refptr_bridge& **find_bridge**(dynamic_id& ID);

This function returns a reference counting pointer to a bridge, after locating it by its identification number.

Refptr_bridge& **find_bridge**(float xpos, float zpos);

This function returns a reference counting pointer to a bridge, after locating it at point (xpos,zpos). This is the first bridge found only.

void **eliminate_bridge**(Refptr_bridge& bp);

This function is used to remove the bridge pointed to by bp from the terrain.

void **remove_bridge**(int i, int j);

This function is used to remove all of the bridges at mapgrid point [i][j].

void **remove_allbridges**();

This function is used to remove all of the bridges from the terrain. Used to reset.

void **bridge_recalc**();

This function is used to recalculate all of the bridges on the terrain.

void **bridge_recalc**(int i, int j);

This function is used to recalculate all of the bridges at mapgrid point [i][j].

bool **bridge_contact**(float xpos, float zpos);

This function returns true if there is a bridge at point (xpos,zpos).

bool **is_def_at_location**(float x, float z);

This function returns true if the bit that represents the area for point (x,z) is on. This is part of the bitgrid that subdivides a gridsquare into smaller squares to reduce the time a vehicle uses the multipoint method of vehicle orientation.

void **set_def_at_location**(float x, float z);

This function turns on the bit that represents the point (x,z) for the presence of a bridge or deformation. Bridges use the same bit because the vehicles use the same method of orientation.

void **add_vehicle**(Refptr_Vehicle& v);

This function adds the vehicle "v" to the terrain. It will put the vehicle on the appropriate mapgrid point linked list of vehicles.

void **remove_vehicle**(Refptr_Vehicle& v);

This function removes the vehicle "v" from the terrain.

void **remove_all_veh**();

This function removes all of the vehicle from the terrain.

void **move_vehicle**(vertex& oldpos,vertex& newpos, Refptr_Vehicle& V);

This function is used when the moving vehicle updates its position. It removes the vehicle from one square's list to the receiving square's list as the vehicle crosses the boundary.

## APPENDIX C PROGRAM USER'S GUIDE

This appendix contains all of the information required to run the dynamic terrain simulation program. The simulator can read terrain from a specified file or create its own terrain. It saves the terrain and the features to a file upon program termination. This appendix will divided into the following sections: How to start the program, and Program Menu and Key Bindings.

### How To Start the program.

The dynamic terrain simulator is start by typing the program name "dynamic" on the command line in the following manner:

% **dynamic** <*M mode* > < *I filename* > < *O filename* >

< *optional*> Definitions

*M* - is the mode of operation for building the terrain, from a file or default.

mode - is N for normal, F for file reading

*I* - indicates that the next argument is the input filename

*O* - indicates that the next argument is the output filename

### Menu Selection and Key Bindings.

**Menu Selections**

**Change Vehicle** - This allows the user to switch to vehicles.

**Grid On** - This allows the user to turn on the grid that outlines the center grid square of the terrain. Useful to find the center of the world and mark the editable terrain elevation points.

**Grid Off** - This turns the center grid square outline off.

**Targeter On** - This turns the feature placement and selector cross hairs on, and must be on to edit or place a feature in the simulator.

**Targeter Off**- This turns the cross hairs off, and disables placement and editing of features.

**Feature Mode** - This selects how the cross hairs are used, Edit or Place.

**Features** - This menu item allows the user to select on of the three features, craters, berms, or bridges, for placement or editing. To edit a feature the appropriate feature type must be selected.

**Materials** - This menu item opens to a list of material types that are available to the user for new features.

**Terrain** - This menu item allows the user to change the appearance of the terrain from an initial flat terrain to terrain with randomly generated elevations, or visa versa.

**Vehicle Dynamics** - This menu allows the user to turn the physically based modeled vehicle dynamics on or off.

**Reset** - This menu item Clears the terrain of all dynamic terrain features.

**Exit** - Exit the program and save the status of the terrain to the default file "outbound" or to the output file specified by the user on program start-up.

## Key Bindings

*Legend:*

| | |
|---|---|
| *KEY* | *Press the key listed.* |
| *KEY1 + KEY2* | *Press both keys together.* |

**MENUBUTTON**- If the right mouse button is pressed the popup menu will appear on the screen.

**LEFTMOUSE** - If the left mouse button is pressed, a dynamic terrain feature will be place or selected, depending on the mode of the targeter and its location.

**LEFT ARROW KEY** - If the targeter is on, it moves left (negative x), otherwise the driven vehicle turns left.

**RIGHT ARROW KEY** - If the targeter is on, it moves right (positive x), otherwise the driven vehicle turns right.

**UP ARROW KEY** - If the targeter is on, it moves up (negative z), otherwise the driven vehicle speeds up.

**DOWN ARROW KEY** - If the targeter is on, it moves down (positive z), otherwise the driven vehicle slows down or goes into reverse, if the speed is 0.

**MINUSKEY** - Decrease the diameter of a editable crater or set the diameter for the next crater to be placed, or decrease the width of a editable berm or bridge.

**MINUS KEY + SHIFT KEYS** - Decrease the height of a editable berm or set the height of the next berm placed.

**MINUS KEY + ALT KEYS** - Decrease the length of a editable berm or set the length of the next berm placed.

**MINUS KEY + F2 KEY** - Decrease the length of ramp one of a editable bridge or set the length of the ramp for the next bridge placed.

**MINUS KEY + F3 KEY** - Decrease the length of ramp two of a editable bridge or set the length of the ramp for the next bridge placed.

**MINUS KEY + F4 KEY** - Decrease the length of the span of a editable bridge or set the length of the span for the next bridge placed.

**MINUS KEY + F5 KEY** - Decrease the height of ramp one of a editable bridge or set the height of the ramp for the next bridge placed.

**MINUS KEY + F6 KEY** - Decrease the height of ramp two of a editable bridge or set the height of the ramp for the next bridge placed.

**PLUS KEY** - Increase the diameter of a editable crater or set the diameter for the next crater to be placed, or decrease the width of a editable berm or bridge.

**PLUS KEY + SHIFT KEYS** - Increase the height of a editable berm or set the height of the next berm placed.

**PLUS KEY + ALT KEYS** - Increase the length of a editable berm or set the length of the next berm placed.

**PLUS KEY + F2 KEY** - Increase the length of ramp one of a editable bridge or set the length of the ramp for the next bridge placed.

**PLUS KEY + F3 KEY** - Increase the length of ramp two of a editable bridge or set the length of the ramp for the next bridge placed.

**PLUS KEY + F4 KEY** - Increase the length of the span of a editable bridge or set the length of the span for the next bridge placed.

**PLUS KEY + F5 KEY** - Increase the height of ramp one of a editable bridge or set the height of the ramp for the next bridge placed.

**PLUS KEY + F6 KEY** - Increase the height of ramp two of a editable bridge or set the height of the ramp for the next bridge placed.

**B KEY** - Stop the driven vehicle.

**RETURN KEY** - Place or select a feature depending on the mode of the targeter and the type of feature selected.

**ALT KEYS + C KEY** - Clears the terrain of all dynamic terrain features.

**ALT KEYS + X KEY** - Exit the program, saving the terrain and features to the default file "outbound" or to the output file specified by the user on start-up.

**ALT KEYS + G KEY** - Toggle the center terrain grid square outline on/off.

**ALT KEYS + T KEY** - Toggle the targeter on/off.

**LEFT BRACKET KEY** - Decrease the elevation of the lower left corner of the center terrain grid.

**LEFT BRACKET KEY + SHIFT KEYS** - Increase the elevation of the lower left corner of the center terrain grid.

**RIGHT BRACKET KEY** - Decrease the elevation of the upper right corner of the center terrain grid.

**RIGHT BRACKET KEY + SHIFT KEYS** - Increase the elevation of the upper right corner of the center terrain grid.

**PAGEUP KEY** - If a bridge has been selected, the mode is edit, and the bridge is destroyed, it is repaired.

**PAGEDOWN KEY** - If a bridge has been selected, the mode is edit, and the bridge is destroyed, it is destroyed.

**DELETE KEY** - If a feature has been selected and the mode is edit, the feature is removed from the terrain.

**HOME KEY** - If a berm or bridge has been selected and the mode is edit, it will change direction in a counter clockwise manner. If the mode is place, the direction of it is being set.

**END KEY** - If a berm or bridge has been selected and the mode is edit, it will change direction in a clockwise manner. If the mode is place, the direction of it is being set.

**V KEY** - This key toggles the users view between inside and outside of a vehicle. Outside the vehicle is the reference point.

**SPACE BALL** - The space ball is used to drive the selected vehicle (Figure 14).

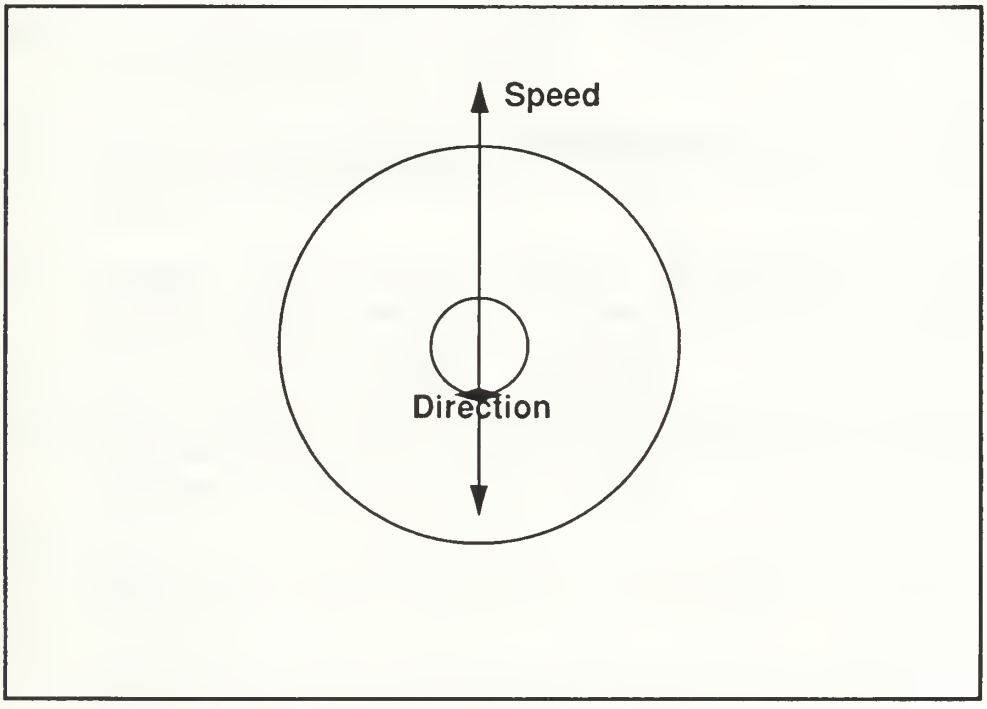



Figure 14: Space Ball

**DIAL BOX** - The dial box is used to control the viewing position when outside of the vehicle (Figure 15).

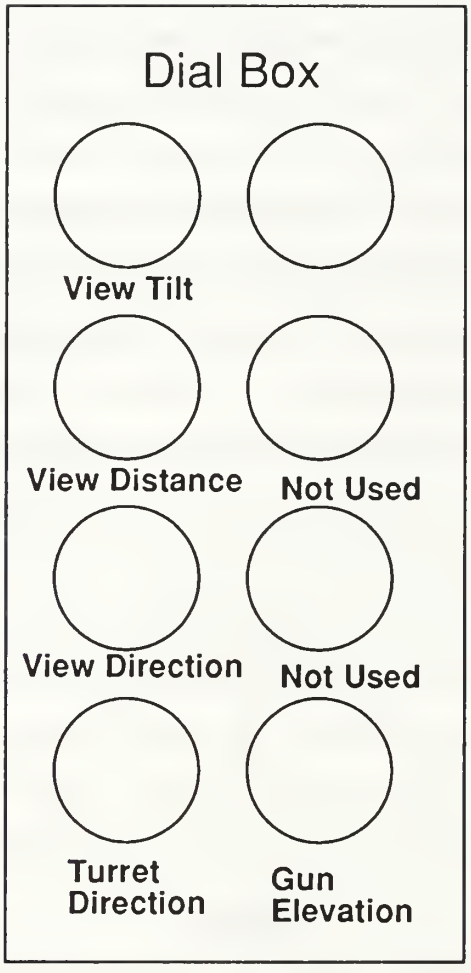



Figure 15: Dial Box

## LIST OF REFERNECES


[ARMY 83] U. S. Army, *Battery Executive Officer's/Platoon Leaders' Handbook, Cannon Artillery*, US ARMY, Field Artillery School, Fort Sill, Oklahoma, p9-3, February 1983.

[IEI 92] IEI Technical Report No. TR-17102-13000-1-05-92, *Design Data Handbook SIMNET/JANUS Interconnection*, Crooks, William H., Fraser II, Robert E., Illusion Engineering, Inc., 28 May 1992.

[IST 91] Institute for Simulation and Training (IST), *Protocol data Units for Entity Information and Entity Interaction in a Distributed Interactive Simulation*, Military Standard (DRAFT), IST-PD-90-2, Orlando, FL, September 1991.

[DMA 86] Defense Mapping Agency Aerospace Center, *Defense Mapping Agency Product Specification for Digital Terrain Elevation Data (DTED)*, Second Edition, April 1986.

[JANU 86] U. S. Army TRADOC Analysis Command, WSMR, JANUS(T) *Documentation Manual*, June 1986

[LATH 92] Latham, Roy, "A Note on Earthworks for Distribributed Simulation", Computer Graphics Development Corporation, Mt. View, CA 5 February 1992.

[LIND 92] Lindberg, Karl, "Dynamic Database Modification in Distributed Simulation", Computer Graphics System Development Corporation, Mt. View, CA, 29 February 1992.

[MOSH 92] Moshell, Michael J., Lisle, Curtis, Blau, Brian, Li, Xin, "Dynamic Terrain Databases For Networked Visual Simulators", Presented at the IMAGE VI Conference, Scottsdale, AZ 14 -17 July 1992.

[PARK 92] Park, H. K., NPSNET: Real-time 3D Ground-Based Vehicle Dynamics, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 1992.

[PRAT 92] Pratt, David R., Zyda, Michael J., Mackey, Randall L., and Falby, John S., "NPSNET: A Networked Vehicle Simulation With Hierarchical Data Structures", Proceedings of IMAGE VI Conference, Scottsdale, AZ. 14 - 17 July 1992.

[WILS 91] Wilson, K.P., "Naval Post Graduate School Class Library - NPSCL - A basic C++ class library", Version 1.0 October 1991

[WILS 92] Wilson, Kalin P., *NPSGDL: An Object Oriented Graphics Description Language For Language For Virtual World Application Support, Master's Thesis*, Naval Postgraduate School, Monterey, CA, September 1992.

[ZYDA 92] Zyda, Michael J., Pratt, David R., Monahan, James G., and Wilson, Kalin P., "NPSNET: Constructing a 3D Virtual World," in Computer Graphics, Special Issue on the 1992 Symposium on Interactive 3D Graphics, MIT Media Laboratory, 29 March - 1 April 1992, pp. 147-156.

## INITIAL DISTRIBUTION LIST

| | |
|---|---|
| Defense Technical Information Center<br>Cameron Station<br>Alexandria, VA 22304-6145 | 2 |
| Dudley Knox Library<br>Code 52<br>Naval Postgraduate School<br>Monterey, CA 93943 | 2 |
| David R. Pratt, Code CS/Pr<br>Computer Science Department<br>Naval Postgraduate School<br>Monterey, CA 93943 | 2 |
| Dr Michael J. Zyda, Code CS/Zk<br>Computer Science Department<br>Naval Postgraduate School<br>Monterey, CA 93943 | 6 |
| LCDR Donald P. Brutzman, Code OR/Br<br>Computer Science Department<br>Naval Postgraduate School<br>Monterey, CA 93943 | 1 |
| Alan K. Walters, LT., USN<br>11312 Penanova St.<br>San Diego, CA 92129 | 2 |